Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Data-Driven Analysis and Modeling of Packet Processing Systems

Sebastian Gallenmüller

Dissertation

Network Architectures and Services NET-2021-02-1

Technische Universität München
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik

Data-Driven Analysis and Modeling of Packet Processing Systems

Sebastian Gallenmüller

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:	1. Prof. Dr.-Ing. Georg Carle
	2. Ass.-Prof. Dr. Gianni Antichi

Die Dissertation wurde am 24.09.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.02.2021 angenommen.

Sebastian Gallenmüller
Data-Driven Analysis and Modeling of Packet Processing Systems
Dissertation, February 2021

Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

ISBN 978-3-937201-71-9
DOI 10.2313/NET-2021-02-1
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)

Network Architectures and Services NET-2021-02-1
Series Editor: Georg Carle, Technical University of Munich, Garching near Munich, Germany

Acknowledgments

This thesis is not the result of the sole endeavor by a single person, but a continuous process supported by many individuals. As I cannot list all of you, the following list is a mere excerpt of the people involved.

First of all, I want to thank my supervisor, Prof. Dr.-Ing. Georg Carle, for the chance to pursue my PhD as a member of your chair. Our nightly discussions were inspirational to many of the ideas and concepts presented in this thesis. Furthermore, I am grateful to my second examiner, Dr. Gianni Antichi, for hosting our research visit to Cambridge and sharing your experience and insights. I also wish to thank Prof. Dr.-Ing. Jörg Ott for chairing the examination committee during the exceptional times of 2021.

Further, I want to thank my fellow colleagues and co-authors Paul, Florian, Daniel, Alexander, Lukas, Torsten, Patrick, Maurice, Stephan, Lukas, Borislava, Wolfgang, Rainer, Maximilian, Dominik, Quirin, Samuele, Fabio, Richard, Fabien, René, Eric, Stefan, Jan, Andreas, Andreas, Henning, Onur, Zenit, Johannes, and Iris.

Lastly, I want to mention the two people who made my academic career possible in the first place, my parents Erich and Ursula. Thank you for your support, financially and emotionally, which allowed me to start this journey. Without your support, I would not be where I am today.

Abstract

The experiment—the pickax of the scientist—is the instrument of choice to mine scientific knowledge. The defining feature of scientific experiments is their reproducibility. Other researchers can recreate experiments to verify or falsify the results of the original experiment. The domain of computer networks rarely reproduces experiments, mostly due to a lack of proper documentation of the original experiment. This thesis proposes a measurement methodology that embeds the concept of reproducibility into experiment design, documentation, and execution.

Our main target of investigation is a two-node setup that allows the benchmarking of packet processing devices. To perform experiments in this setup, we propose an experiment workflow that relies on full automation for experiment configuration, execution, and evaluation, thereby creating repeatable experiments. By releasing the experiment artifacts, including the experiment scripts, the measured data, and their analysis, other researchers can either try to replicate the results of the original experiment or even create their own experimental setup to reproduce the original results.

Further, we present measurement tools that allow the effective investigation of the packet processing devices. An essential prerequisite of network experiments is the precise definition and replay of the network traffic at the input of the investigated device. Therefore, we analyze different packet generators and develop suitable packet generation strategies. To enable an accurate delay analysis, we use hardware-generated timestamps.

The measurement methodology and the measurement tools are applied to two different domains: high-performance packet processing applications and wireless networked control applications. The former domain typically relies on specialized packet processing frameworks. We analyze the frameworks themselves and other applications based on them, such as software routers or intrusion prevention systems. In contrast to traditional kernel-based network processing, these frameworks offer a ninefold increase in throughput that almost scales linearly with the number of available processing cores. At the same time, service quality rises. The latency and especially the worst-case behavior improve when relying on such a high-performance framework. The requirements and the conditions of the second application domain differ significantly. Wireless links are lossy and sensitive to interference, poor preconditions for control applications where low latency and loss rates are crucial. An additional challenge is the execution of repeatable measurements due to the highly sensitive network links. We present an approach using a shielded environment for repeatable wireless network experiments. Furthermore, we create our own platform for control systems consisting of a robot and a benchmarking suite to perform realistic measurements for this domain.

Models can help quantify the observations of the presented measurements and describe the behavior of packet processing systems. The thesis presents a novel modeling technique to predict the performance of packet processing devices. It analyzes the components of packet processing devices, such as the CPU or system buses, and their individual performance. The performance of the components is then combined to predict the overall performance of the packet processing device. We check the validity of our models by applying them to the performed measurements.

Zusammenfassung

Das Experiment — die Spitzhacke des Wissenschaftlers — ist das Werkzeug der Wahl zur Förderung wissenschaftlicher Erkenntnis. Ein entscheidendes Merkmal wissenschaftlicher Experimente ist deren Reproduzierbarkeit. Andere Wissenschaftler können Experimente wiederholen, um Ergebnisse des ursprünglichen Experiments zu bestätigen oder zu widerlegen. Die wissenschaftliche Disziplin der Informatik reproduziert nur selten Experimente, meist weil die ursprünglichen Experimente nur unzureichend dokumentiert sind. Im Rahmen dieser Arbeit schlagen wir eine neue Messmethodik vor, die das Konzept der Reproduzierbarkeit zu einem festen Bestandteil von Experimentdesign, -dokumentation und -ausführung macht.

Hauptgegenstand der Untersuchung ist ein Messaufbau, bestehend aus zwei Knoten, der es erlaubt, Benchmarks von Paketverarbeitungssystemen durchzuführen. Zur Durchführung der Experimente in diesem Messaufbau schlagen wir einen vollautomatisierten Arbeitsablauf für Konfiguration, Durchführung und Auswertung der Experimente vor, um so wiederholbare Experimente zu generieren. Durch eine Veröffentlichung aller Bestandteile eines solchen Experiments, wie der Experimentskripte, der gemessenen Daten und ihrer Analyse, können andere Wissenschaftler versuchen, entweder die Ergebnisse des ursprünglichen Experimentes zu replizieren oder mittels eigenem Messaufbau zu reproduzieren.

Außerdem stellen wir Messwerkzeuge vor, die eine effektive Untersuchung von Paketverarbeitungssystemen erst ermöglichen. Eine wesentliche Voraussetzung für Netzwerkexperimente ist die präzise Beschreibung und Wiedergabe des Netzwerkverkehrs als Eingabe für ein untersuchtes System. Dazu analysieren wir unterschiedliche Paketgeneratoren und entwickeln geeignete Strategien zur Erzeugung von Netzwerkverkehr. Zur genauen Analyse von Latenzen verwenden wir hardwaregenerierte Zeitstempel.

Messmethodik und -werkzeuge werden auf zwei unterschiedliche Anwendungsbereiche angewandt: hochperformante Paketverarbeitungsprogramme und drahtlose vernetzte Regelungssysteme. Erstere verwenden typischerweise spezialisierte Paketverarbeitungsframeworks. Wir untersuchen die Frameworks selbst und darauf basierende Programme, zum Beispiel Softwarerouter oder Intrusion-Prevention-Systeme. Im Gegensatz zu traditioneller kernelbasierter Netzwerkverarbeitung, bieten die Paketverarbeitungsframeworks einen um den Faktor 9 gesteigerten Durchsatz, der nahezu linear mit der Anzahl der Prozessorkerne skaliert. Der Einsatz eines hochperformanten Paketverarbeitungsframeworks steigert auch die Servicequalität, so wird das Latenzverhalten auch unter größter Belastung verbessert. Die Anforderungen und die Bedingungen des zweiten Anwendungsbereiches unterscheiden sich fundamental. Drahtlose Verbindungen sind verlustbehaftet und störungsempfindlich, schlechte Voraussetzungen für Regelungssysteme, die auf niedrige Latenzen und Verlustraten angewiesen sind. Eine zusätzliche Herausforderung ist die Durchführung wiederholbarer Messungen aufgrund der störungsempfindlichen Verbindungen. Wir zeigen einen Ansatz für wiederholbare drahtlose Netzwerkexperimente mit Hilfe einer abgeschirmten Umgebung. Darüber hinaus erstellen wir unsere eigene Plattform für Regelungssysteme, die aus einem Roboter und einer dazugehörigen Benchmarksuite besteht, um realistische Messungen in diesem Anwendungsbereich durchführen zu können.

Modelle können dabei helfen die Beobachtungen der vorgestellten Messungen zu quantifizieren und das Verhalten der Paketverarbeitungssysteme zu beschreiben. Diese Arbeit stellt eine neuartige Modellierungstechnik zur Performanzvorhersage von Paketverarbeitungssystemen vor. Sie analysiert die Komponenten von Paketverarbeitungssystemen, wie zum Beispiel die CPU oder Systembusse und deren jeweilige Performanz. Die Performanz der einzelnen Komponenten wird dann kombiniert, um die Performanz des Gesamtsystems vorherzusagen. Wir überprüfen die Validität der so gewonnenen Modelle anhand der durchgeführten Messungen.

Contents

Chapter 1

Introduction

2^{32} or 4.3 billion is the theoretical number of possible addresses of the Internet Protocol version 4. The fact that this address space does not suffice to address all devices connected to the Internet demonstrates the magnitude of today's computer networks. In view of this sheer size and complexity, we decided on a slightly less ambitious target of investigation: a two-node network.

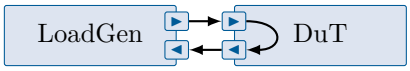


Figure 1.1: A two-node network consisting of a load generator (LoadGen) and a device under test (DuT)

Figure 1.1 depicts a typical setup of such a network. There, a load generator transmits traffic to the DuT, which processes the received packets and forwards the resulting traffic back to the load generator. By observing ingress and egress traffic on the load generator, the behavior and the performance of the DuT can be determined. However, this radically simplified network setup is not the result of an aversion against complex systems, but the attempt to create an isolated, controlled environment. Experiments in such a network allow studying the effects of the DuT unaffected by other nodes. RFC 2544 [1] defines a benchmarking methodology for network interconnect devices based on such a setup. Despite dating back to 1999, the fundamental goals of network benchmarking are still relevant today.

This thesis follows the spirit of RFC 2544, utilizing the well-established recommendations of network performance benchmarking. The proposed two-node setup, for instance, is used throughout this thesis with only minor additions. The benchmarking methodology itself is updated where necessary to reflect the technical progress for network devices and measurement equipment, which has taken place since. Besides the raw bandwidth increase, novel device architectures have emerged, processing packets entirely in software or hardware with tightly integrated software components. The result is a radically changed device behavior that led to the design of high-precision and high-performance measurement tools. Furthermore, this thesis presents an updated methodology and the necessary tools to perform network benchmarks reproducibly.

Network experiments in this thesis focus on two main areas: high-performance packet processing systems and networked control systems (NCS). Both areas feature unique requirements; therefore, the benchmarking methodology is extended for each area individually. The experiment results lay the foundation to devise a modeling framework. This framework incorporates the experiment results to deduce models being able to predict the performance of packet processing systems with a particular focus on high-performance systems and NCS.

1.1 Research Questions

The thesis is shaped along the following research questions:

- RQ1: How can we design and execute reproducible experiments for the investigation of packet processing systems?
- RQ2: How can we create a measurement methodology to identify the main impact factors on packet processing performance?
- RQ3: How can we create a modeling framework to efficiently and adequately describe the behavior of packet processing systems in general?
- RQ4: How can we characterize, analyze, and model high-performance packet processing systems?
- RQ5: How can we characterize, analyze, and model wireless networked control systems?

The following paragraphs define the framework of the research topics covered in this thesis. The proposed research questions highlight relevant areas of network experiments. They lead the way towards our contribution to advance the area of applied network experiments with a focus on reproducibility and modeling.

RQ1: How can we design and execute reproducible experiments for the investigation of packet processing systems? The network experiments in this thesis typically consider a two-node setup consisting of a load generator and a DuT. The ultimate goal is the creation of reproducible experiments in such a setup that allows other researchers to create the same results reliably over many executions. An initial step towards this goal is a reliable repetition of one's own experiments. Crucial to this is the ability to observe and record the state of the entire experiment. Without the means to observe all the relevant aspects of the investigated system, an experiment cannot be recreated successfully. We consider different aspects as relevant. Before the start of an experiment, the initial configuration of the entire setup must be recreated. This setup involves the hardware and software for load generator and DuT. During the experiment, the entire process should be executed the same way like any other run, e.g., using identical measurement intervals or traffic patterns, to ensure reliable result recreation. We include the evaluation of results in our experiment. By processing the measurements identically across different test runs, we additionally provide a consistent and, therefore, repeatable evaluation of the experiments.

Chapter 2 introduces different stages of reproducibility and defines a methodology focused on the creation of replicable network experiments. We present our testbed based on the plain

orchestrating service (pos), which offers a workflow that supports researchers to design such experiments ranging from device setup and configuration over execution to the final evaluation. The workflow relies on a high degree of automation to avoid human influence and hence its impact on the results of an experiment.

RQ2: How can we create a measurement methodology to identify the main impact factors on packet processing performance? Employing a testbed and a fully automated workflow allows network experiments covering a large parameter space. The values of the parameters and the combination of different experimental parameters can be varied for every execution. The number of experiments grows exponentially with the number of parameters for the combinations. This growth puts an upper limit to the searchable parameter space that can be sensibly explored using experimental evaluation. Our goal is the selection of the parameters that have the most substantial impact on the performance of packet processing.

Technical progress leads to an ever-increasing performance for network devices. To investigate such powerful devices, equally capable measurement tools are required. Our first requirement is the development of tools that allow the measurement of the network device output even for high-load scenarios. These tools enable the effective identification of high-impact parameters. The second requirement is the accurate generation of the input for a given experiment. Our tools must be able to create the input according to the researcher's specification to enable the recreation of experimental results.

In Chapter 3, we introduce the packet generator MoonGen that can be used as a reliable source for packet generation. Further, MoonGen's recording capabilities are presented that allow the detailed analysis of delay caused by the investigated network devices. With Ethernet further developing towards 100G and higher bandwidths, we also present tools that leverage specialized hardware and optimized data structures to support future high-bandwidth network experiments.

RQ3: How can we create a modeling framework to efficiently and adequately describe the behavior of packet processing systems in general? Beside identifying the main impact factors on packet processing performance, we want to gain insight into the connection between these impact factors and the resulting performance. Therefore, we create models that can describe these connections to predict the performance of packet processing systems. Models of this kind allow for the design of packet processing systems performing a specified task at an expected performance level. The over- or underdimensioning of packet processing systems leads to unnecessarily high costs due to idling resources or overloaded components. By avoiding these costs, correct predictions can help to design and build efficient systems.

Another important aspect of modeling is the scalability of packet processing tasks. Network traffic can be processed independently—typically on a per-packet or on a per-flow basis—which allows spreading the processing tasks across independent processing units. This independence allows for efficient multi-core scaling of packet processing tasks. Our models can help to describe how efficiently packet processing systems scale for specific workloads on a given target.

Packet processing systems are constructed from smaller components. Each of these subcomponents has its own capacity, which may ultimately limit the performance of the overall packet

processing system. Models can help to understand how the interaction of these individual bottlenecks determines the performance of the entire system. Knowing these models can help direct development efforts towards these bottlenecks unlocking the full performance of packet processing systems.

Chapter 4 introduces a modeling framework to describe the performance of packet processing tasks based on available system resources. This resource model is used in the following chapter to describe the behavior of high-performance packet processing systems. Measurements are also used to validate the performance predictions of the model with real measurements.

RQ4: How can we characterize, analyze, and model high-performance packet processing systems? Traditional system architectures were equipped with a single CPU, packet processing happened in the kernel, and applications used the socket API for network communication. New developments, such as new Ethernet standards with higher bandwidths or new offloading features, were fully transparent for the network applications. This transparency meant that the underlying architecture of network applications remained unchanged for Ethernet bandwidths up to 1 Gbit/s.

When shifting to 10G Ethernet, the traditional architecture could not cope with the increased performance. This shortcoming led to the creation of specialized high-performance packet processing frameworks, such as the Data Plane Development Kit (DPDK) or netmap, that allow utilizing high bandwidths. These frameworks employ radically changed architectures, which were explicitly designed for modern multi-core systems and bypass traditional in-kernel packet processing and network APIs to increase performance. Fundamental changes result in a changed behavior for packet processing applications, which requires a reevaluation and further analysis of such systems. Increasing the performance further impacts the hardware usage. System buses, which were sufficient for bandwidths of 1 Gbit/s, may become relevant bottlenecks for higher bandwidths. We explore the handling of packet IO in parallel, which was not possible or relevant for traditional architectures. Modeling can help to describe the performance of these new packet processing frameworks and predict their limitations.

Chapter 5 investigates several high-performance packet processing applications: frameworks for packet processing, software routers, and intrusion prevention systems. We analyze the throughput and latency behavior of the investigated systems and provide models to predict their performance.

RQ5: How can we characterize, analyze, and model wireless networked control systems? Besides high-performance packet processing systems for wired networks, there are also applications relying on wireless systems that operate on bandwidths of several Mbit/s or lower. Control systems are such an application domain, where a remote controller is connected wirelessly to a mobile entity that requires reliable, regular inputs to perform critical control tasks. For such systems, reliable service with stable and low latency is essential. The amount of data to transport is limited. Therefore, the raw throughput figures are less important in such a use case. The behavior of WLAN significantly differs from Ethernet. Wireless networks have higher error rates that impact the reliability of the packet transfer in general and retransmission schemes in WLAN

may increase the delay. Suitable experiments are needed to cover these aspects relevant to the requirements of WLANs and wireless NCS.

The shared nature of the wireless medium makes it highly susceptible to interference and other external factors. A testbed for wireless network experiments must take these factors into account to gain repeatable experimental results. In addition, the DuTs can be resource-constraint systems that limit the possibilities for measurements on the DuT without impacting performance.

The shared medium requires complex access mechanisms to allow successful data transmissions across wireless networks. Access control schemes, resource-constraint systems, and environmental conditions impact WLAN performance. Adequate models must be designed to reflect these systems and the requirements of WLAN. Modeling must also consider that WLAN does not rely on specialized frameworks but kernel network stacks and traditional kernel IO interfaces, influencing network performance.

Chapter 6 applies our measurement methodology to the application domain of NCS. We chose a wireless NCS as a target for our investigation. Therefore, we introduce a platform called NCSbench consisting of a balancing robot and a corresponding control stack. Further, we present a benchmarking suite focused on replicable experiment execution and present an evaluation between replicated experiments. A testbed is created to perform repeatable network experiments using pos.

1.2 Outline

Chapter 2 introduces our measurement methodology for reproducible network experiments and the testbed designed around this methodology. Measurement tools and their analysis are covered in Chapter 3. In Chapter 4, a modeling technique is presented that allows performance predictions based on a bottleneck analysis of current packet processing systems. Chapter 5 presents measurements focused on high-performance packet processing systems, which typically involve servers handling network traffic of 10G Ethernet and above. The investigation of wireless packet processing systems is the focus of Chapter 6. There, we apply our measurement methodology to wireless NCS. Chapter 7 summarizes the contributions of this thesis and discusses open research questions.

Chapter 2

Measurement and Benchmarking Methodology

This chapter introduces our measurement methodology—the underlying frameset applied to the experiments conducted throughout this thesis. It defines the fundamental terminology and the necessary performance measures to report experimental results. We present network experiments and benchmarks to characterize the behavior of packet processing systems adequately. Therefore, we created an experiment workflow to design, execute, and evaluate experiments with a particular focus on reproducibility.

2.1 Terminology and Key Performance Indicators

Section 2.1 is based on a collaboration between Daniel Raumer, Sebastian Gallemüller, Florian Wohlfart, Paul Emmerich, Patrick Werneck, and Georg Carle [2].

We define a *network experiment* as a process where we investigate a DuT in a computer network, such as the two-node network in Figure 1.1. During this process, *measurements* are performed to determine and record the state and the behavior of the investigated network device. From measurement results, *key performance indicators (KPI)* are distilled to report the performance of a device compactly. In a *benchmark*, network experiments determine the performance of network devices.

RFC 2544 [1] defines four basic KPIs:

- *Throughput:* the maximum rate a DuT can process without packet loss.
- *Latency:* the processing time of a DuT measured from reception on the ingress to the transmission on the egress port.
- *Loss rate:* the rate of dropped packets during the steady state of a DuT.
- *Back-to-back frames:* the maximum number of packets in a burst that a DuT can process without dropping packets.

This thesis conducts measurements of these KPIs for high-performance systems and NCS in Chapters 3, 5, and 6: throughput benchmarks, to measure the performance of different systems, loss rates, to determine if and when a DuT is overloaded, and back-to-back frames, to demonstrate that device behavior changes for bursty traffic patterns. Measurements of the three previously mentioned KPIs were performed following RFC 2544. For latency measurements, the RFC proposes the reporting of a single latency value that represents the average of 20 measurements. This average value is not sufficient to describe the network behavior used for critical NCS or high-performance packet processing systems. Modern measurement equipment allows the extensive measurement of latency values. The additional amount of information enables further analysis, such as the determination of worst-case behavior or variance in latency.

2.2 Reproducible Network Experiments

Sections 2.2, 2.3, and 2.4 are based on a collaboration between Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [3], the presented experiment artifacts are part of joint work between Sebastian Gallenmüller, Johannes Naab, Iris Adam, and Georg Carle [4].

This section presents a methodology for reproducible network experiments. We explain how this methodology is embedded into our own testbed and our self-developed testbed controller, called plain orchestrating service (pos). Measurements performed in our testbed that follow the pos methodology can lead the way towards the creation of reproducible network experiments.

2.3 Related Work

Independent reproduction of results is vital to understanding and validating scientific results. A full reproduction is aided not only by resulting measurement data, which is occasionally published along with scientific publications but also by including raw measurement data and configuration descriptions. This additional data may contain tools and scripts used to configure, run, and evaluate the experiment. An ACM policy [5] considers reproducibility as a three-stage process, with full experiment reproduction being the final stage:

1. **Repeatability** is achieved if the *same* team using the *same* experimental setup can reliably recreate their results.
2. **Replicability** is accomplished if a *different* team using the *same* experimental setup can recreate the original results.
3. **Reproducibility** requires a *different* team utilizing a *different* experimental setup that can recreate the initial results without reusing the original artifacts.

The ACM developed a series of badges that can be awarded to papers that successfully demonstrated replicability or reproducibility. The implementation of badging into the review process is considered a favorable way towards reproducible research [6], [7]. Bajpai et al. [8] suggest guidelines and tools to conduct reproducible network experiments. All previously mentioned

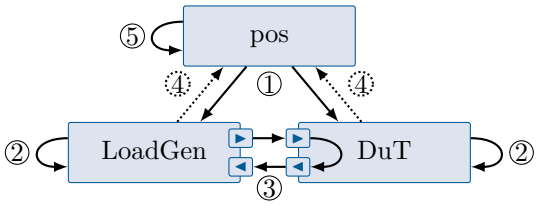


Figure 2.1: Experiment workflow using the pos testbed controller (cf. Gallenmüller et al. [3])

measures target reproducible network research in general; in this thesis, we focus on specific experiments. We investigate small-scale network experiments running on real hardware, e.g., in testbeds.

Nussbaum [9] compares different testbeds and demonstrates how their functionalities, such as the automated configuration and experiment execution, can be used to create repeatable experiments. However, these testbeds were not built to run experiments involving distributed nodes with more complex network topologies. Zilberman conducts a case study [10] demonstrating that even papers awarded with the reproducibility badge may not paint a complete picture of the investigated system behavior. She observed low robustness, i.e., minor variations of the original input, such as the investigated packet size, could lead to a substantially different outcome.

We designed our testbed from scratch with reproducibility in mind. Following our experiment workflow leads to the creation of repeatable experiments. Further, we optimized the testbed to run the two-node benchmarking setups with full control over the hardware. We rely on fixed, non-switched wiring to avoid any influence of external components on the measured network setup. The complete automation of the experiment workflow tries to address the issue of low robustness. Experiments can be run for different configurations with low additional effort for the researcher to reduce the number of blind spots not covered by experimental results.

2.4 Testbed for Reproducible Network Experiments

For our network experiments, we set up a testbed with a focus on testing packet processing systems. This testbed consists of different servers equipped with commodity hardware and 1G, 10G, and 40G Intel or Mellanox network interface cards (NICs) (e.g., I350, X520, X540, X710, XL710, and ConnectX-4 Lx EN). Figure 2.1 depicts a typical experiment configuration and workflow in our testbed. A minimal example of our testbed topology consists of a two-server setup (LoadGen, DuT) and an orchestrating server (pos). The entire experiment workflow is controlled by pos running on a separate server that deploys (1), configures (2), executes (3), and collects (4) the measurement artifacts of network experiments, before automatically evaluating (5) the measured data. As we use live images for our experiments, the configuration state is lost after a system reboot. This loss of state enforces testbed users to use scripts for configuring experiment servers and the subsequent evaluation. At first glance, this burdens users and might be considered a disadvantage; however, it has three major benefits:

1. Using live images, our experiment hosts start the experiments from a well-defined state. In addition, the testbed user must include the system configuration into the experiment scripts. The full automation of the pos experiment workflow creates repeatable experiments and minimizes the chance of accidental misconfiguration.
2. The testbed can be accessed remotely, which allows easy access to our testbed for other research groups. Experiment results can be recreated by members of these research groups, thereby creating a replicable experiment.
3. Scripts used for experiment configuration, execution, and evaluation document the entire experiment workflow. Releasing these scripts together with the experimental results can provide other research groups with the foundation to perform their own experiments. The pos experiment workflow cannot create fully reproducible experiments but support a convenient workflow towards reproducibility.

While our testbed can be accessed remotely, we restricted access to known and trusted persons (members of our research group and research collaboration partners). We need to rely on trust, as we provide root access to all testbed machines often required for network experiments. The remote access enables others to replicate our testbed experiments. Thereby, our experiments reach the second stage for reproducibility without any additional effort by the researcher. We call this property replicability by design. Achieving the third stage, reproducibility, cannot be achieved by relying on the testbed and its processes but instead needs other scientists to take up the challenge of creating the same results utilizing their tools and test equipment. Unfortunately, the pos experiment workflow cannot guarantee this kind of reproducibility by design. However, the experiment artifacts, including the setup, execution, and evaluation scripts, provide enough information for other scientists to develop their own experiments to reproduce the initial experimental results. Therefore, pos supports a path towards a reproduction of results by other research groups. Grosvenor et al. [11] demonstrate an easily accessible way to publish the results of their paper. Every figure of their paper links to a website containing experiment setup, description, and results. We followed their example by utilizing the pos artifacts for one of our publications [4].

Our test setup allows for both black-box tests and white-box tests. For typical black-box tests, data is collected on the egress and ingress ports of the load generator and used to determine metrics, such as throughput and latency. White-box tests are also possible by recording the behavior on the DuT itself, for instance, by profiling the interrupt rate of NICs or the cache load caused by applications. Our automated testbed can record many features in parallel, leading to gigabytes of data for tests running only a few minutes. The data generated in the testbed can be processed, plotted, and used to derive accurate models.

2.5 Limitations

Network experiments depend on the topology of the investigated network. We are typically interested in measuring the behavior of our DuT directly without any influence of other interconnecting devices, such as switches or routers. Therefore, all setups in this thesis use networks with

direct non-switched connections. Using direct connections has the disadvantage that topologies cannot be (re)created automatically and require the researcher to wire the network topology physically. There exist optical L1 switches that allow linking fibers optically, which add a constant delay offset due to the internal wiring of the switch. The impact on the forwarding delay of such a switch is lower than 15 ns [12], which is considerably lower than an L2 cut-through switch with approximately 300 ns [13]. Such a setup would allow automating the topology with a predictable low impact on delay. However, due to the high costs, the testbed is not equipped with such an optical switch.

The recreation of results is currently limited to configurations accessible from the operating system (OS). However, there may be configurations influencing packet processing performance, such as BIOS settings or NIC firmware. Setting these configurations via pos would be possible. However, BIOS configurations or flashing firmware differs across different manufacturers. Due to this lack of standardized configuration interfaces, pos does currently not support automated configuration. For this thesis, we rely on default settings for BIOS and firmware where possible and specify where the configuration was changed, e.g., in Section 5.4.

2.6 Key Results

This chapter introduces basic terminology, KPIs, a testbed, and its design principles for creating repeatable and replicable network experiments. It further introduces pos, which employs these design principles automating the experiment configuration, execution, and evaluation. Automation still has its limits when it comes to controlling firmware and its configuration with its non-standardized interfaces. What makes pos and the underlying methodology unique for academia is its focus on experiment publication. Following the proposed experiment structure, pos simplifies the preparation of the experiment artifacts for publication, as demonstrated for one of the author's publications [4]. The availability of experiment artifacts allows others to recreate their own experiments, thereby achieving reproducible network experiments.

2.7 Author's Contributions

Section 2.1 is based on a collaboration between Daniel Raumer, Sebastian Gallemüller, Florian Wohlfart, Paul Emmerich, Patrick Werneck, and Georg Carle [2]. The author significantly contributed to the analyses of network KPIs in this paper.

Sections 2.2, 2.3, and 2.4 are based on work by Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [3]. The author contributed to the development of the methodology and the implementation of pos and the testbed. The artifacts [4] mentioned were created and published by the author.

Chapter 3

High-Performance Measurement Tools

According to specification, 10G Ethernet can transmit approximately 14.88 Mpkts/s. To effectively benchmark DuTs supporting 10G Ethernet, measurement tools must be equally powerful, if not more powerful, due to additional measurement tasks. This chapter presents various tools enabling high-performance network experiments of systems that can handle millions of packets per second.

3.1 Motivation

The motivational example in Figure 3.1 and the following analysis is based on joint work by Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle [14].

Figure 3.1 demonstrates how the behavior of high-performance software packet generators can impact the results of network experiments. There, a two-node setup was used with Open vSwitch (OvS) in Version 2.0.0 on Debian Linux (kernel v3.7.9) using a 3.3 GHz Intel CPU. Traffic is generated with MoonGen using rates between 0 and 2 Mpkts/s with a packet length of 64 B. Timestamps are recorded utilizing hardware timestamps on an Intel 82599 NIC. Figure 3.1 demonstrates how the median forwarding latency of OvS behaves if the inter-packet gap is varied between experiments. The baseline performance with a burst size of 1 is created using constant bitrate (CBR) traffic, i.e., the inter-packet gap between consecutive packets is constant. The same measurement has been repeated several times with bursty traffic patterns, i.e., several packets are put onto the wire back-to-back with appropriately longer gaps in between bursts. Figure 3.1 shows the relative deviation between the CBR at $x = 0$ and the bursty traffic patterns. Even when ignoring the low-load and high-load scenarios, the median latency almost doubles compared with the baseline scenario.

To increase efficiency, packet processing frameworks work on batches, i.e., they process several packets per function call. High-performance packet processing frameworks default to batch sizes of up to 512 packets. Packet generators relying on these frameworks can create bursty traffic.

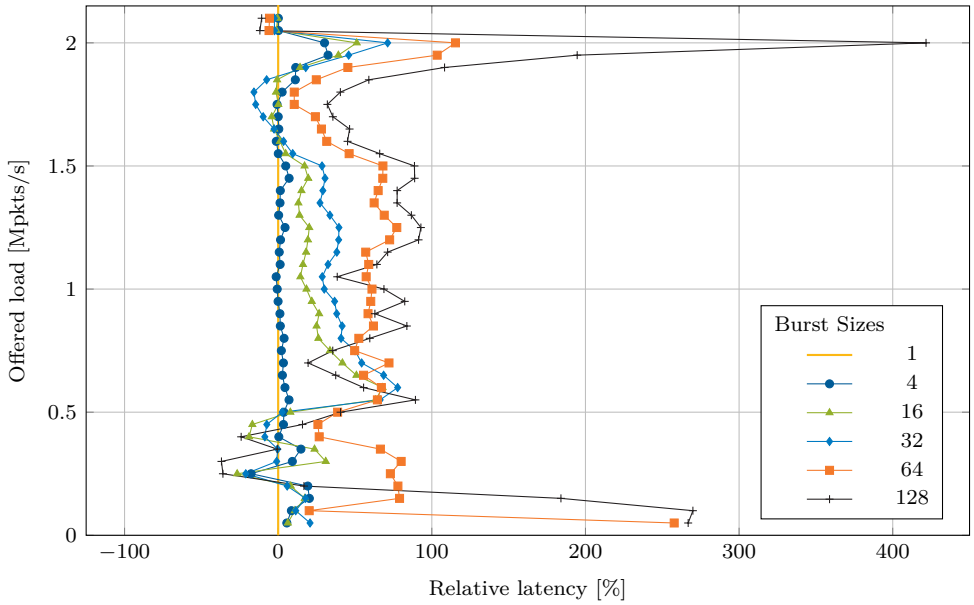


Figure 3.1: Relative deviating latency for measurement traffic with different burst sizes (cf. Emmerich et al. [14])

Figure 3.1 shows that this parameter can have a significant impact on the latency performance of a DuT. Therefore, the type of traffic used must be documented for every experiment. Additionally, the packet generators must be able to control the amount and characteristics of the generated traffic precisely, to neither overload the DuT nor to measure unwanted side-effects of the generated traffic. This section presents high-precision measurement tools developed for the packet rates of 10G Ethernet with 14.88 Mpkts/s or more.

3.2 MoonGen

Section 3.2 is based on work by Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [3]; on a collaboration between Sebastian Gallenmüller, Paul Emmerich, Daniel Raumer, and Georg Carle [15]; and on a publication by Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle [16].

MoonGen [16] is a high-performance, open-source software packet generator based on the high-speed packet processing framework DPDK. MoonGen can generate minimum-sized packets at a rate of 10 Gbit/s (14.88 Mpkts/s) using a single core with packets generated by user-defined Lua scripts.

Figure 3.2 shows the architecture of MoonGen. DPDK offers access to the NICs, with libmoon on top of it, providing a convenient and straightforward user API. MoonGen is realized as a libmoon application. Originally, libmoon has been part of MoonGen, but in 2016, MoonGen was refactored to provide two separate frameworks—a generic packet processing framework and a dedicated packet generator [3]. User-programmable scripts run on top of MoonGen and

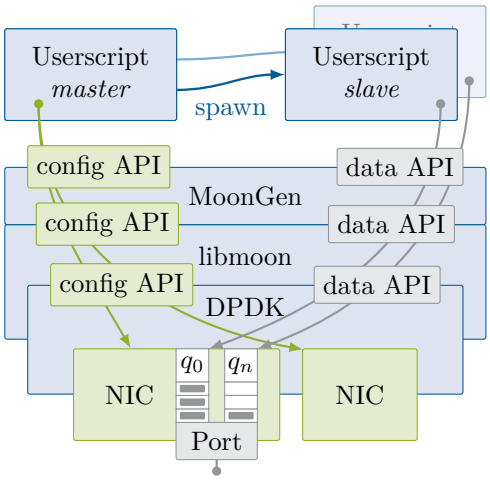


Figure 3.2: Architecture of MoonGen/libmoon (cf. Gallenmüller et al. [3])

perform the packet processing and measurement tasks. There are two different kinds of scripts: *master userscripts*, which perform configuration tasks and spawn the *slave userscripts* that perform the actual processing tasks. User scripts are written in the Lua language [17], an easy-to-learn scripting language ideal for rapid prototyping. To improve performance, MoonGen uses the LuaJIT [18] compiler. Furthermore, LuaJIT includes a foreign function interface (FFI) to conveniently and efficiently embed existing C code, such as DPDK's own libraries. The most relevant features of MoonGen applied in this thesis are its capabilities for high-precision traffic generation and hardware timestamping.

Timestamping: One important feature is the hardware-assisted timestamping of packets, which allows delay measurements in the sub-microsecond range. The timestamping resolution is high enough to measure the cable lengths of an optical fiber utilizing a time-of-flight measurement [15]. There exist two different timestamping implementations in MoonGen:

The first timestamping implementation [16] misuses registers on the Intel 10G NICs (e.g., 82599 and X540) that were originally intended for the precision time protocol (PTP). Misusing this hardware feature comes with certain limitations concerning the type and the number of packets that can be timestamped. First, PTP can either run directly on Ethernet or UDP. Therefore, timestamping packets must either use the PTP Ethertype or UDP. In addition, UDP PTP increases the minimum packet size to 80 B. Second, to account for the drift between two network interface ports, even if located on the same NIC, PTP clocks are reset before each timestamp measurement. This restriction limits the number of timestamped packets to one packet in flight per round trip. Assuming an end-to-end latency of 1 ms this approach can timestamp approximately 1 kpkts/s.

The second timestamping implementation in MoonGen uses the hardware capabilities of the Intel X550 NIC [19], which can append receive timestamps to the packet buffers. This feature does not come with any limits concerning the type or the number of packets to timestamp. However, timestamping is only possible for received but not for sent packets. This limitation

can be circumvented by adapting the measurement setup. An example of such a setup is given in Section 5.4.3.

The advantage of the first implementation is its simple two-node setup and support across different Intel 82599 and X710-based chips. A disadvantage is the necessity to change the timestamped packets and the limited number of timestamped packets per second. The second implementation requires a more complex setup but allows timestamping arbitrary traffic at line rate. This second method is suitable to observe rarely happening latency events, whereas the first method with its simpler setup is preferable in situations where a median latency suffices.

High-precision traffic generation: Controlling the inter-packet gap is key to reliably generating different specific traffic patterns, such as CBR or bursts. MoonGen offers three methods for controlling pattern generation [14]:

1. NICs such as the Intel 82599 and X710 offer hardware capabilities for rate control, which we call *hardware-supported approach* [20], [21]. These cards offer a possibility to limit the transmit rate, effectively creating a possibility to generate CBR traffic.
2. The *pure software approach* tries to control the packet rate by precisely timing the handover of packets from software to the NIC. However, NICs fetch the packets asynchronously from RAM with Direct Memory Access (DMA), PCIe, and NIC buffers impacting precision.
3. The third approach fills the inter-packet gap with invalid packets to determine the transmission time of a valid packet on byte-level. Packets are invalidated using wrong frame check sequences. Hence, we call it *corrupt CRC approach*. This approach may influence the performance of a DuT. However, the frames with wrong checksums are typically dropped early in the processing path, e.g., on the receiving NIC itself, which has no impact on the overall performance of the investigated packet processing task. This assumption may not hold for all devices. Therefore, we recommend testing the corrupt CRC approach for each DuT individually before relying on this approach exclusively.

MoonGen implements all three approaches, whereas other software packet generators only implement the pure software approach. An in-depth analysis follows, which compares the quality of packet generation for the mentioned three approaches and other state-of-the-art software packet generators.

3.3 Analysis of Software Packet Generators

Section 3.3 is based on work by Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle [14].

Multiple software packet generators are available that were specifically designed for Ethernet with bandwidths of 10 Gbit/s or more (see Table 3.1). To allow repeatable measurements, we require a tool that allows a high degree of control over the generation process to allow specific traffic patterns and a reliable creation thereof. Therefore, we analyzed the quality of different software packet generators along the following criteria:

	Version	IO API
Pktgen-DPDK [23]	v2.8.0	DPDK (v1.8.0)
MoonGen [24]	git 5cf96c72*	DPDK (v1.8.0)
pkt-gen [25]	git b24fce99	netmap
pfq-gen [26]	v5.2.9	PFQ
zsend [27]	v6.3.0.160209	PF_RING ZC

*) Used for CBR traffic and hardware timestamping

Table 3.1: Investigated software packet generators (cf. Emmerich et al. [14])

1. **Bandwidth:** How fast is a packet generator in terms of packets per second?
2. **Accuracy:** How close is the average observed rate to the configured one?
3. **Precision:** How much do individual inter-packet gaps deviate from the configured value?

Figure 3.1 depicts the results of an experiment where only bandwidth and accuracy were considered. The bursty traffic patterns represent different experiment results for a high deviation of the inter-packet gap, i.e., a low precision. In contrast to CBR traffic, the bursty traffic has highly asymmetric inter-packet gaps. Within a burst, packets are sent back-to-back, i.e., an inter-packet gap of 0. In between bursts, the inter-packet gap is maximized to meet the configured rates. To reliably recreate experiment results, the inter-packet gap must be specified and the packet generator must offer high precision, i.e., it can create the correct inter-packet gaps.

Experiment Setup

The setup for the following experiments uses a two-node setup with the DuT being the investigated software packet generator and Open Source Network Tester (OSNT) [22] being the receiving node. OSNT utilizes the NetFPGA 10G platform and can timestamp packets with a resolution of 6.25 ns. The DuT is equipped with an Intel i7-960 CPU (3.2 GHz base frequency) and an Intel X520 NIC (Intel 82599 Ethernet controller) running Ubuntu Linux 14.04 LTS (kernel 3.16).

Table 3.1 presents the high-performance software packet generators to be evaluated. Every generator relies on high-performance packet processing frameworks. The frameworks netmap, PFQ, and PF_RING ZC have included their own packet generators as a part of their example applications. Pktgen-DPDK and MoonGen are built on top of DPDK. For a fair comparison, two versions of the packet generators were selected, which use the same version of the underlying DPDK. Currently, MoonGen is the only packet generator supporting the hardware-assisted and the corrupt CRC approaches for packet generation. Therefore, MoonGen experiments are repeated thrice to evaluate the available packet generation approaches.

Section 5.1 contains an in-depth analysis of the frameworks DPDK, PF_RING ZC, and netmap.

Evaluation: Bandwidth, Accuracy, and Precision

Table 3.2 shows that all of the investigated packet generators can create millions of packets per second. Even the generator with the lowest packet rate, pfq-gen, was still able to generate 5.67 Mpkts/s with default settings. All packet generators were able to accurately match the

Packet Generator	Batch Size (Default)	Throughput (Default) [Mpkts/s]	Throughput (Precise) [Mpkts/s]
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

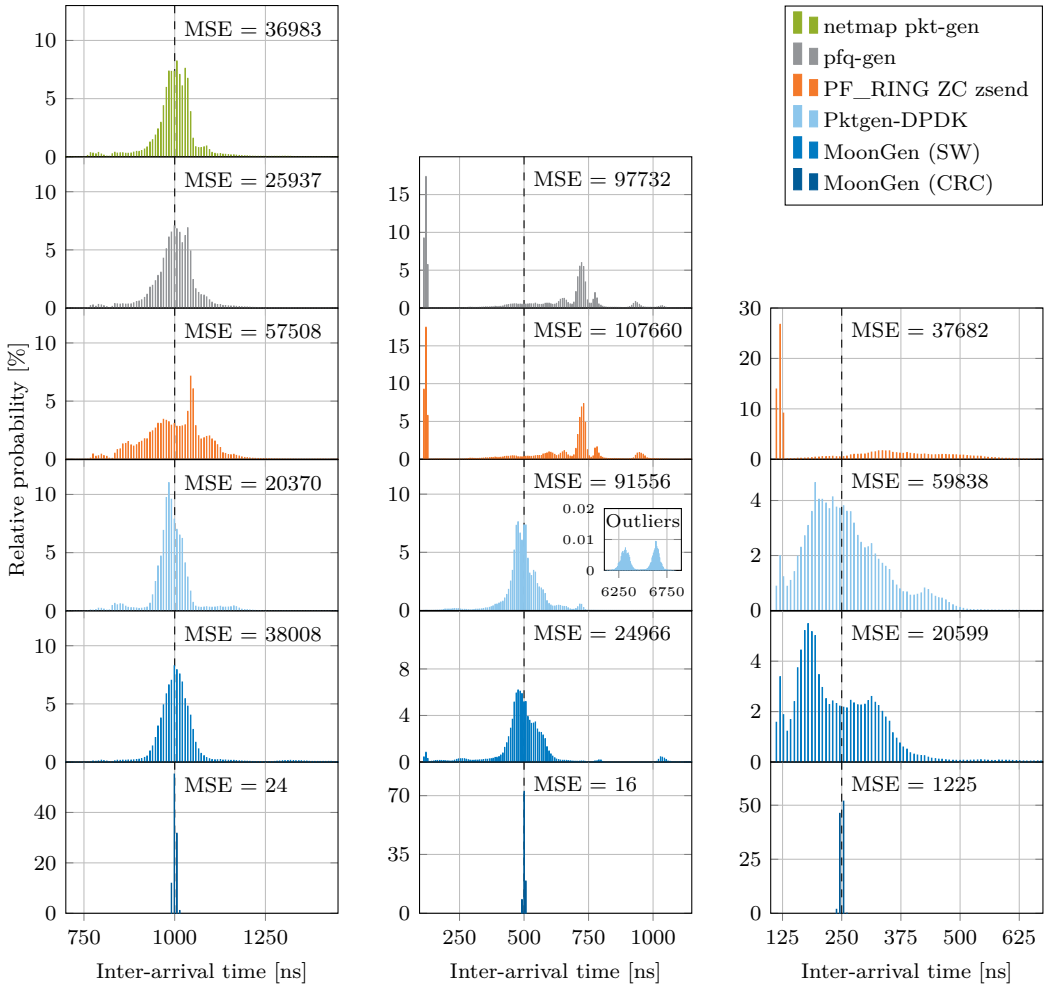
[1] Intel 82599, highest reliable hardware setting
[2] No imprecise generation possible
[3] Not precise at high rates despite configuration

Table 3.2: Packet rates of packet generators optimized for maximum throughput and high precision (cf. Emmerich et al. [14])

configured packet rates as long as they were not overloaded with two exceptions. The hardware-supported approaches had a precision error of up to 3.3 % (Intel X710). The rate control on these NICs is intended to limit the traffic of different VMs sharing a common host [21]. For such a use case, a lower precision or bursty traffic behavior seems acceptable. Pktgen-DPDK failed to match the configured rates due to a software bug. All of the investigated packet generators can deliver the bandwidth and accuracy for testing 10G Ethernet devices.

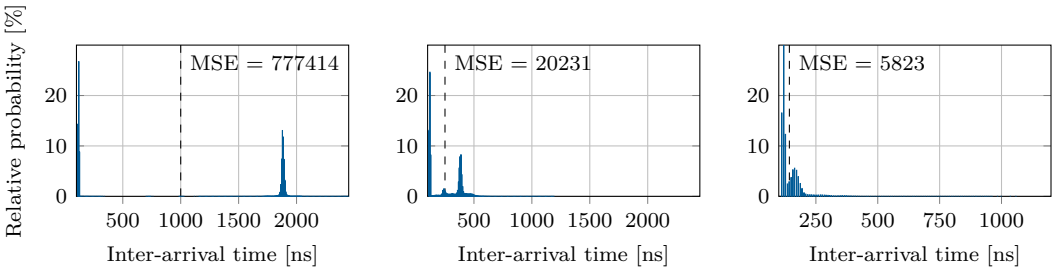
Table 3.2 shows the batch sizes used in their default settings. Due to the undesired effects of bursty traffic generation, we repeat our measurements with the highest precision settings for each packet generator. These settings lead to a decreased performance for all investigated packet generators. In contrast to the other frameworks, netmap relies on costly system calls for sending packets [28], which is the reason for its high decrease in performance. Table 3.2 further presents the maximum throughput of precise traffic generation but does not evaluate the quality of the precision. Therefore, we evaluate the inter-packet gaps while generating CBR traffic. CBR traffic requires a constant inter-packet gap. Higher rates require shorter gaps leading to a twofold effect: generating higher packet rates and timing gaps precisely without creating bursty traffic become more challenging. As the precise settings, for the non-hardware-assisted approaches, do not allow line rate bandwidth (cf. Table 3.2), we run the following tests with a packet size of 128 B to test high packet rates with high bandwidth.

Figure 3.3 shows the distribution of the inter-packet gaps as histograms, the mean squared error (MSE) of the inter-packet gap is added as an indicator of precision. Ideally, for CBR traffic, the inter-packet gaps would all be mapped to a single bucket matching the expected inter-packet gap leading to an MSE of 0. Figure 3.3a shows the performance of the packet generators at a rate of 1 Mpkts/s. PF_RING ZC zsend offered the lowest precision, which we attributed to flawed timekeeping in its architecture. The corrupt CRC approach of MoonGen offered the highest precision. Its distribution is within the timer granularity of OSNT. In Figure 3.3b, the rate is increased to 2 Mpkts/s. The packet generator of netmap was not able to generate this rate. For the other packet generators, the histograms get wider and the MSE increases, except for MoonGen's corrupt CRC approach. PFQ and PF_RING ZC have similarly shaped histograms indicating a common root cause for their performance. Despite their different architectures, they



(a) Packet rate of 1 Mpkts/s (b) Packet rate of 2 Mpkts/s (c) Packet rate of 4 Mpkts/s

Figure 3.3: Inter-packet gap of packet generators (cf. Emmerich et al. [14])



(a) Packet rate of 1 Mpkts/s (b) Packet rate of 4 Mpkts/s (c) Packet rate of 7 Mpkts/s

Figure 3.4: Inter-packet gap of hardware-assisted generation on Intel 82599 (cf. Emmerich et al. [14])

share a single common component—an adapted version of the ixgbe driver—which we assume to be the root cause. For the DPDK-based packet generators, shipping their own drivers, we did not observe such a histogram shape. We observed high-valued outliers for Pktgen-DPDK being the reason for its reduced precision. Despite sharing the DPDK framework with Pktgen-DPDK, MoonGen's software generation approach did not suffer from these high outliers. Figure 3.3c displays the performance at a rate of 4 Mpkts/s. The histogram shapes and the MSE improve compared with the previous measurement for the purely software-based approaches. However, in this measurement, all three packet generators create bursty traffic with the lowest measured bucket representing back-to-back frames. The wrongly assumed improvement happens as the expected inter-packet gap is closer to a bursty traffic pattern than before, so generating a bursty pattern is less penalized than in the previous measurements. The measurement also shows that the corrupt CRC approach has its limits. The high MSE value is caused by high outliers in the μs-range when starting MoonGen.

Figure 3.4 shows the results of the hardware-assisted approach of rate control. The 82599 chip, despite its high performance, proved to be imprecise. It generated bursts of two for any configured rate. These bursts create a bimodal distribution, with roughly half of the packets sent back-to-back and the other half of the packets having an inter-packet gap roughly doubled compared with the expected value.

High MSE values indicate the imprecision (cf. Figures 3.4a and 3.4b). The hardware-assisted approach enables higher rates than the other two approaches. This imprecision becomes less relevant with higher rates, indicated by a lower MSE (cf. Figure 3.4c). Therefore, the hardware-assisted approach delivers a reasonable precision at rates not achievable using the alternative approaches.

Influence of CPU microarchitectures

Packet generation is a highly demanding task for the CPU. This section explores the impact of both CPU microarchitecture and clock speed on this process. The comparison investigates two packet generators: pfq-gen and MoonGen. We selected the latter as an example of the DPDK userspace driver against one with a patched kernel driver as found in PFQ, netmap, and PF_RING ZC.

Pkt. Gen.	CPU [GHz]	Throughput [Mpkts/s]
MoonGen	3.5	12.9
	1.6	6.8
pfq-gen	3.5	7.2
	1.6	3.6

(a) Impact on performance

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

(b) Impact on precision of MoonGen (SW) (2 Mpkts/s)

Table 3.3: Impact of CPU frequency on packet generation (cf. Emmerich et al. [14])

Micro Architecture	CPU [GHz]	Throughput [Mpkts/s]	MSE [ns^2]
Ivy Bridge	1.60	1	29382
		2	29318
		3	19628
Ivy Bridge	3.50	1	26818
		2	19133
		3	15237
Nehalem	3.46	1	28700
		2	24218
		3	15239

Table 3.4: Generation rates at different clock frequencies (cf. Emmerich et al. [14])

Impact of CPU frequency on generation rate: Table 3.3a shows the impact of clock frequency on the generation rate for pfq-gen and MoonGen (pure software approach for rate control). We use the pure software approach for MoonGen, representing the most challenging approach from a CPU perspective, as the CPU cannot rely on hardware or corrupted CRCs for precise packet timing. The experiment is executed on an Intel Xeon E3-1265L v2 with a maximum frequency of 3.5 GHz with turbo-boost and repeated with the CPU manually throttled to 1.6 GHz. Both packet generators react similarly: the throughput scales well with the CPU frequency. These results are expected as high-performance IO frameworks are known to scale linearly with the CPU frequency (cf. Section 5.1). Therefore, we consider the linear scaling as a property of the underlying frameworks rather than a property of the packet generators themselves.

Impact of CPU frequency on precision: The following tests concentrate only on the MoonGen traffic generator, as it proved to be the most precise at rates of 2 Mpkts/s and above. We configure the software packet rate control to generate 2 Mpkts/s to quantify how the CPU clock frequency influences the precision. Previous measurements show this rate is well below the generation limit for any clock frequency, i.e., enough processing cycles are available to cope with the task. Despite the availability of enough clock cycles to fulfill the task, a difference in the MSE of the packet distribution is visible in Table 3.3b. The error decreases as the CPU frequency increases, i.e., faster CPUs achieve higher precision, even if the additional CPU cycles are not required to generate the desired rate.

Impact of CPU microarchitecture on precision: The third investigation involves two different CPU microarchitectures: Ivy Bridge released in 2012 (Intel Xeon E3-1265L v2) and Nehalem

released in 2008 (Intel Core i7-960). We generate 1, 2, and 3 Mpkts/s with MoonGen's software rate control and measure the precision as MSE in Table 3.4. As with the previous results in Figure 3.3, the MSE improves for the software rate control with increasing rates, as burst traffic patterns are penalized less at higher rates. There is only a small difference between the analyzed microarchitectures when clock frequencies are almost identical. The clock frequency has a more substantial impact on the precision than the microarchitecture itself.

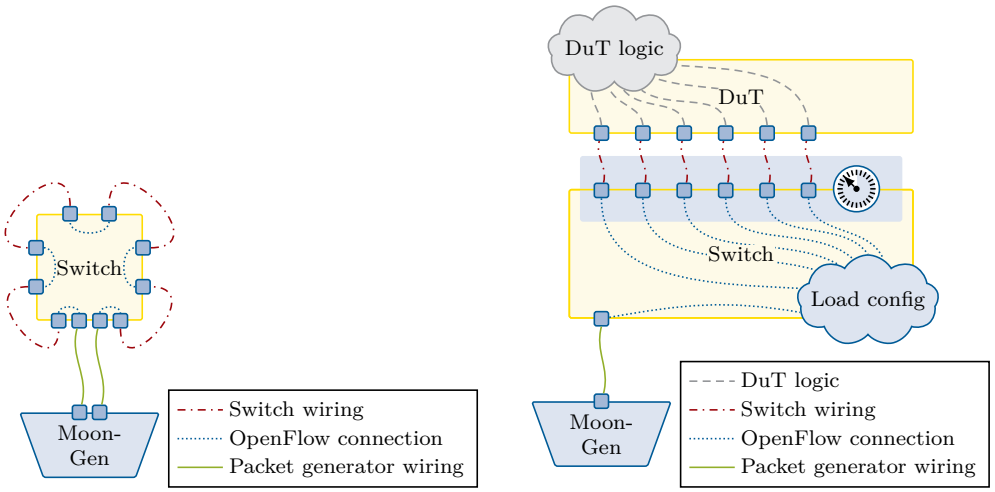
The CPU-related measurements show the importance of the clock frequency for performance and precision. Additionally, there is only a minor influence of the microarchitecture on the precision, at least the investigated ones. The higher the CPU frequency, the better performance and precision. With lowering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

3.4 FLOWer

Section 3.4 is based on a publication by Paul Emmerich, Sebastian Gallenmüller, and Georg Carle [29].

Regular server hardware offers a limited number of NIC ports, restricting the bandwidth of software packet generators. Testing switches with 32 or more ports requires a potent and costly load generator setup to test the switch at full bandwidth capacity. Hardware generators such as NetFPGA/OSNT face the same problem. FLOWer [29] solves this problem by combining a regular load generator platform with a programmable switch to create a cost-efficient measurement platform with extended bandwidth and measurement capabilities. FLOWer relies on MoonGen as a packet generator. However, the concept is not limited to a specific packet generator. In 2018, Ramanujam et al. [30] presented a similar concept, the Simple Network Tester, based on OSNT.

Figure 3.5a shows a setup configuration with a single switch. There, two ports of a switch are attached to a load generator running MoonGen and the remaining switch ports are connected to other ports. Previous, similar setups either used broadcast messages [31] or VLAN forwarding rules [32] to test the bandwidth or power consumption of the switch. FLOWer uses OpenFlow to configure the switch allowing more sophisticated benchmark scenarios. One of the scenarios is quality of service (QoS), where MoonGen creates two types of traffic flows—realtime and background traffic. The background traffic is amplified using the OpenFlow flood action to create a high load on the switch, while MoonGen timestamps the realtime traffic forwarded through the switch between MoonGen's interfaces. Figure 3.5a demonstrates how the OpenFlow configuration creates a single cycle through the switch for the realtime traffic. The length of this cycle can be adapted by either increasing or decreasing the number of petals to measure the forwarding latencies between the different ports. FLOWer was applied to an Edge-Core Networks AS5712-54X 10 GbE running PicOS (using 48× 10G Ethernet ports). The MoonGen



(a) Single-switch self test setup

(b) Two-switch amplification setup

Figure 3.5: Switch measurement setups of FLOWer (cf. Emmerich et al. [29])

server was equipped with a Xeon E3-1230 v2 with a dual-port Intel X520-T2 NIC. With QoS enabled, we measured a forwarding latency between 1 and 3.5 μs for the realtime traffic. If background traffic was increased to a value above 10 Gbit/s, latency remained at a level of 3.5 μs. Benchmarking with different cycle lengths, we measured a constant forwarding latency of approximately 729 ns per 10G port.

In the single-switch setup, the switch—the DuT—additionally performs tasks of the load generator, such as the amplification of the background traffic. This amplification task can impact the performance of the DuT. The two-switch setup displayed in Figure 3.5b divides the tasks between two switches. One switch is a dedicated DuT and the second switch prepares the traffic, eliminating any potential impact of the benchmarking task on the DuT. OpenFlow rules can be used to amplify and alter the traffic received from MoonGen to create separate flows. Further, OpenFlow meters on the benchmarking switch can be used to collect traffic statistics or to limit the amount of traffic passed to MoonGen. Latency measurements can be performed using MoonGen by forwarding the timestamp packets through the benchmarking switch. However, the benchmarking switch increases the measured latency and may cause additional jitter. The used switch introduced a forwarding latency of 729 ns and jitter of up to 218 ns.

3.5 FlowScope

Section 3.5 is based on a collaboration between Paul Emmerich, Maximilian Pudelko, Sebastian Gallenmüller, and Georg Carle [33] and on joint work by Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [3].

FlowScope is a tool to record and analyze packet dumps for network debugging and network forensics. Established tools fail at recording or analyzing bandwidths of 100 Gbit/s. FlowScope can capture at a rate of 120 Mpkts/s even with 128-byte packets when using multiple threads.

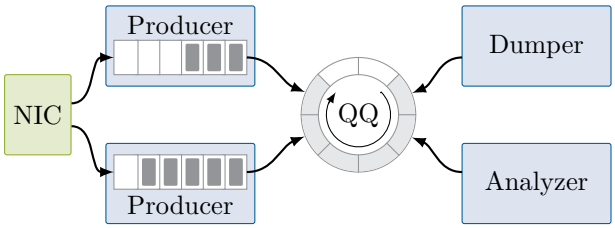


Figure 3.6: Architecture of FlowScope and the QQ data stucture (cf. Gallenmüller et al. [3])

Bro Time Machine [34], a similar tool, managed to process a packet rate of 2.4 Mpkts/s, a 50-fold performance decrease compared with FlowScope.

FlowScope utilizes Receive Side Scaling (RSS) and multi-queues leading to an efficient multi-threaded design. However, the key to its performance is founded in the novel in-memory data structure, organized as a ring buffer, illustrated in Figure 3.6. The elements of this ring buffer (or outer queue) are queues that contain the actual packets. This structure of queues within queues coins QQ, the data structure's name. QQ supports the multi-producer/multi-consumer scheme. Figure 3.6 depicts the producers on the left and the consumers on the right-hand side. Every producer has exclusive access to one of these inner queues for recording packets, which renders explicit synchronization redundant. After an adjustable amount of recorded data or a specified timeout, the producer stops filling its inner queue. This inner queue is handed over to the outer queue and the producer begins filling a new inner queue. After releasing an inner queue, it can be consumed by one of two different processes—the analyzer or the dumper. The analyzer allows peeking at packets without removing them from the queue. Using libpcap-based filter expressions, an analyzer can trigger a dumper process that dumps selected packets to disk. This operation removes the packet from the queue. Access to the outer queue is rare and hence handled with locks to facilitate multiple accessors. A lock-based outer queue allows implementing special features such as the time-traveling dumper process that would be hard to implement in a fully lock-free queue. An inner queue is only handed out to a single producer or consumer and does not need locks. This design choice allows us to maintain a high performance despite using locks at the high-level interface, which is accessed rarely.

Whereas QQ offers unprecedented throughput figures, it introduces a considerable amount of latency. Packets can only be read after a producer hands its inner queue over to the outer ring of queues. The latency originates from the desired size or specified timeout of the inner queues. Considering other impact factors on latency, such as the expected data rate or the number of concurrent producer threads, latencies in the range of up to several hundred milliseconds are possible. This delay is a significant increase compared with other queues operating in the range of several hundred nanoseconds [35].

3.6 Key Results

We demonstrated that the packet generator's quality has a significant impact on the outcome of a network experiment and thus its repeatability. Therefore, we created MoonGen, our own

packet generator, that allows a high degree of control over the packet generation process, offers bandwidths of up to 10 Gbit/s with minimum-sized packets on a single core, and measures latency in hardware in the ns-range. An analysis of current software packet generators demonstrates that MoonGen's generation quality exceeds the abilities of other software packet generators. To increase generation performance, we further introduce the FLOWer concept, combining MoonGen with a programmable switch. We leverage the capabilities of the programmable switches for the purpose of a packet generator. This setup allows us to benchmark devices, such as switches, which we could not perform with the limited number of ports on a software packet generator. Current state-of-the-art network analyzers fail at processing packet rates of 2.4 Mpkts/s. By trading delay for bandwidth, FlowScope enables the investigation of network traffic at 120 Mpkts/s.

3.7 Author's Contributions

Section 3.2 is based on work by Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle [16]. The author contributed to the description, the illustrations, and the analysis of MoonGen. A demonstrator of MoonGen—a joint publication between Sebastian Gallenmüller, Paul Emmerich, Daniel Raumer, and Georg Carle [15]—presents a setup to measure cable lengths and describes MoonGen's basic features. The author contributed to the implementation of the demo, its analysis, and its description. He contributed significantly to the second timestamping implementation, which is first described in this thesis.

Section 3.3 is based on work by Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle [14]. The author contributed to the measurements presented in this paper, focusing on the impact of the CPU on performance.

Section 3.4 is based on a publication by Paul Emmerich, Sebastian Gallenmüller, and Georg Carle [29]. The author contributed to the analysis, description, and illustration of FLOWer.

Section 3.5 is based on work by Paul Emmerich, Maximilian Pudelko, Sebastian Gallenmüller, and Georg Carle [33]. The author co-supervised the thesis, in which FlowScope was implemented, contributing to the ideas and measurements presented.

High-level descriptions of MoonGen, libmoon, FLOWer, and FlowScope are contained in a publication by Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [3]. The author contributed to the description and analyses of the mentioned applications.

The author created the architecture illustrations in this chapter. Measurement plots are joint work with the respective co-authors.

Chapter 4

Modeling Framework

Several models are created and used throughout this thesis for different applications and scenarios. The models in this thesis are based on a common modeling framework. This chapter provides the technical background for this modeling framework.

Figure 4.1 depicts a high-level view of a packet processing system s investigated in this work. The investigated system itself is created from different subcomponents—hardware and software—resulting in a complex packet processing system. Exposed to a vector of input parameters I, this packet processing system behaves in a certain way creating a vector of output parameters O. Typical examples for input parameters of packet processing systems are the available bandwidth, the traffic used for testing, or the configuration of the investigated packet processing system. The outputs of the packet processing system can be measured, for instance, the packet losses or the delay caused by packet processing. A mathematical model of this packet processing system is expressed in Equation 4.1:

$$s(I) \mapsto O \tag{4.1}$$

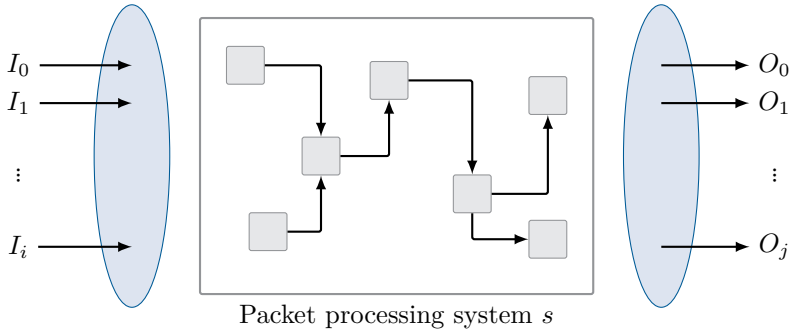


Figure 4.1: Generic packet processing system

Equation 4.1 describes a complete model of the whole system, i.e., all possible input parameters generating all possible output parameters. However, such a model comes at a price. A high number of input parameters n of a model increases the costs for obtaining the required parameters, tracking them, and their subsequent processing. Calculating all output parameters may also be unnecessary as only a subset of the output parameters is required in a particular situation. Therefore, this work focuses on creating efficient models predicting the output required for a specific use case. We select the models considering the following goals:

Parsimony: An obvious factor to evaluate the quality of a model is the deviation between its prediction and the actual measurement. However, the exclusive focus on the prediction error may lead to complex and expensive models. Information criteria were created to consider additional properties for the quality estimation of models. Two common information criteria are the Akaike information criterion (AIC) [36] and the Bayesian information criterion (BIC) [37]. A particular feature of AIC and BIC is their attention to the number of model input parameters. Both, AIC and BIC, penalize additional input parameters, thereby preferring simpler models. We consider two advantages. First, simpler models help to explain and communicate the interconnections of complex systems on a high level. Second, simplicity can foster the adoption of models in applications. With a lower number of input variables to track, model implementation becomes easier, execution faster, and calculation cheaper. Following the parsimony principle regarding the number of input parameters, we aim for models that require a low number of input parameters.

Expressiveness: The overall performance of a packet processing task can be described as the combined performance of the involved subcomponents. Our measurements show (cf. Chapters 5 and 6) that typically a single subcomponent presents the bottleneck limiting the overall performance of the packet processing task in an investigated scenario. For a packet processing task, the available Ethernet bandwidth is such a possible bottleneck. As long as no other component is overloaded, the overall throughput performance is limited by this bandwidth. In such a case, a model can ignore the performance of all the other subcomponents. This simple example demonstrates that the creation of simple models with a limited number of input parameters is possible—the goal of parsimony can be reached without compromising their expressiveness.

Measurability: Measurements can be divided into two distinct classes, black-box and white-box measurements. Black-box measurements can be obtained by observing the inputs and outputs of a DuT. Being non-intrusive, the DuT is not impacted by measurement tasks, which leads to more accurate measurement results. Additionally, black-box tests are simpler to execute as a DuT can be easily integrated into existing measurement setups because of the highly standardized IO interfaces of packet processing systems. White-box tests, executed on the DuT itself, can lead to additional insight into the packet processing task. However, white-box measurements can impact the performance of a DuT. On-device measurements require additional effort to adapt the measurement capabilities to a specific DuT. Additionally, vendors may restrict access to the system making measurements more difficult or even impossible. In this work, we prefer measurements obtainable via black-box testing due to the universal applicability

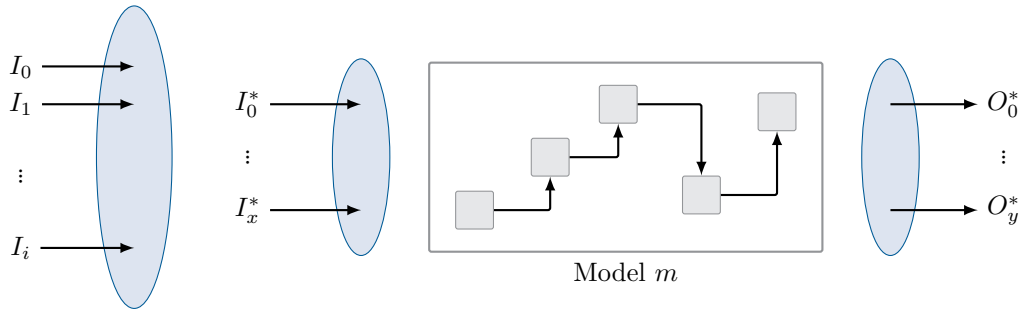


Figure 4.2: Generic model for packet processing systems

and simplicity. White-box testing is used to investigate assumptions of the underlying system's behavior, deduced from black-box measurements.

A model m of system s is shown in Figure 4.2. The modeled system m does not use the entire input vector I, but a subset of it we denote as I^*. The selection of input parameters I_x^* for model m is made according to the previously introduced goals: we optimize for a minimal number of input parameters (parsimony), we remove parameters that are not relevant for a specific use case (expressiveness), and we prefer input parameters that can be determined easily (measurability). Model m further describes only a subset of output parameters O^*, which are relevant for a specific use case, out of the original output parameters O. At the same time, all modeled output parameters in O^* should match their measured counterparts in O as closely as possible. The model is described by Equation 4.2:

$$m(I^*) \mapsto O^* \tag{4.2}$$

4.1 Analysis of Software Packet Processing Systems

Section 4.1 is based on joint work by Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle [38], this publication itself is based on the Master's Thesis [39] by the author.

In the following, we want to analyze typical software packet processing systems. Figure 4.3 visualizes a path of a packet through a typical off-the-shelf system. After arriving at a NIC, packets are transferred via PCIe either to memory via DMA or using Intel Data Direct I/O (DDIO) [40] directly into the last level cache (LLC) of the respective CPU. Despite being attached directly to the CPU, DMA and DDIO work asynchronously. This asynchrony means that packets, from the perspective of the CPU cores, become available for processing without actively dedicating CPU cycles to reception. Sending packets utilizes the same asynchronous techniques as reception without active CPU involvement. Asynchronous transfer helps to reduce the CPU cycles spent on packet processing as the application only dedicates cycles to the packet processing task after the packet has been placed in RAM or LLC. The actual packet processing task is executed on the CPU cores after the packets have been placed in RAM or LLC. On multi-CPU systems,

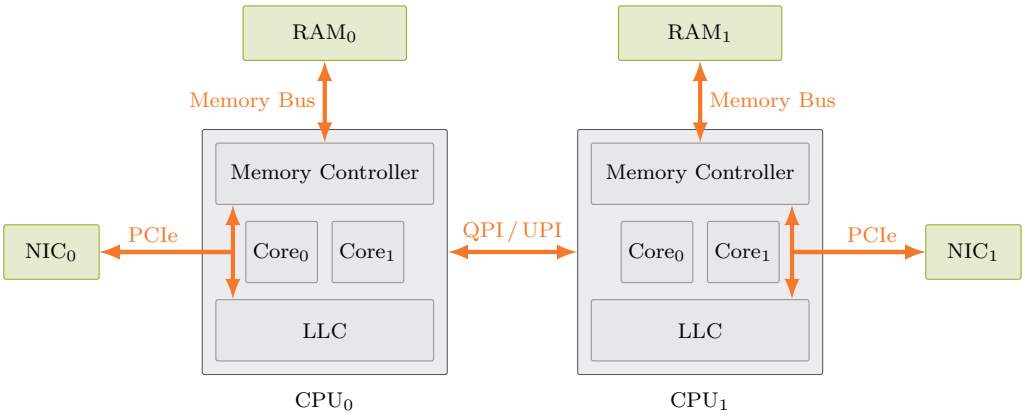


Figure 4.3: System architecture of software packet processing sytems

Interconnect	Duplex	Maximum Bandwidth [Gbit/s]	Maximum Packet Rate [Mpkt/s]
10G Ethernet	full	10	14.88
40G Ethernet	full	40	59.52
100G Ethernet	full	100	148.81
PCIe 2.0 8× [43]	full	32	
PCIe 3.0 8× [43]	full	63	
PCIe 4.0 8× [43]	full	125	
DDR3-800 [44]	half	51	
DDR3-1600 [44] / DDR4-1600 [45]	half	102	
DDR4-3200 [45]	half	205	
QPI [41]	full	77	
UPI [42]	full	166	

Table 4.1: System interconnect bandwidths

depicted in Figure 4.3, packet processing may involve the CPU interconnect, such as Quick Path Interconnect (QPI) or Ultra Path Interconnect (UPI) [41], [42]. Packets use this interconnect if they are received or sent via a NIC attached to a remote CPU.

This system architecture presents two crucial resources to the packet processing task: the interconnect bandwidth between individual components and the CPU time available on the CPU cores. The packet processing task can be limited if either of these resources run out. In our model m, each of the different system interconnects or CPU resources are represented by their own input parameter I_x^*. We aim for a low number of input parameters while keeping the model accurate. Therefore, we analyze the potential bottlenecks in the following to identify the relevant input parameters for modeling.

4.1.1 Interconnect Bottlenecks

The hardware possesses hard limitations such as the maximum bandwidth of Ethernet, PCIe buses, or the RAM, which can act as a bottleneck for packet processing systems. These bottlenecks act as strict upper bounds which cannot be exceeded. Table 4.1 lists different limits

that were relevant to the systems investigated in this thesis. The table contains the maximum bandwidth in one direction, i.e., full-duplex capable connections can transmit the given rates in both directions simultaneously.

There are two limiting factors for Ethernet, the raw bandwidth determined by the Ethernet standard and the throughput in packets per second. For the 10G Intel 82599, we did not observe any limitations independent of the packet rate. The 40G Intel XL710 has hardware restrictions regarding the maximum packet rate. We measured a maximum packet rate of 33.7 Mpkts/s, i.e., a minimum packet length of 128 B is required to reach a bandwidth of 40 Gbit/s [33]. At the time of writing, we did not have access to 100G hardware; therefore, hardware limitations are still unknown.

PCIe offers different bandwidths depending on the version and the number of physical lanes allocated. PCIe offers a point-to-point connection between the NIC and the CPU, i.e., the bandwidth is not shared across multiple PCIe devices. For 10G, the PCIe 2.0 with 8 lanes is sufficient even for a dual-port NIC. However, for 40G, the PCIe 3.0 with 8 lanes does not suffice to support a NIC with two ports at line rate. The Intel XL710 is available in such a configuration with PCIe 3.0 and 8 lanes being the maximum supported configuration [21]. For 100G, only the newer PCIe 4.0 with at least 8 lanes can support the needed bandwidth for a single port.

The main memory standards relevant for this thesis are DDR3 and DDR4, each offering many different standardized bandwidths. Table 4.1 contains only the minimum and maximum standardized rates for both versions using a single memory channel, respectively. Modern Intel CPUs offer between 2 (Sandy Bridge microarchitecture) and 6 (Skylake microarchitecture) memory channels. If each of the memory channels is equipped with memory modules, the available bandwidth increases by a factor of 2 to 6 [46]. Even the lowest bandwidth of 51 Gbit/s is enough to support multiple 10G ports per system. For 40G and 100G, the memory bandwidth can have an impact depending on the number of NIC ports and memory channels in use. In contrast to Ethernet or PCIe, the memory bandwidth is not a hard limit. Intel's DDIO [40] technique allows NICs to copy the packets directly into the LLC. The transfer of packets into and out of LLC reduces the pressure on the memory bus. The bandwidth of these DDIO-accelerated systems depends on factors such as the LLC usage of other running tasks. Intel claims an IO performance of approximately 250 Gbit/s for a DDR3-based system [40].

For systems with multiple CPUs, the CPU interconnect can act as a bottleneck. QPI was introduced in 2008 and offered an initial bandwidth of 77 Gbit/s. The interconnect bandwidth was updated several times; the latest version of the CPU interconnect from 2017 is called UPI that roughly doubled the bandwidth compared with the initial version of QPI. The CPU interconnect is used by packets forwarded between NIC ports attached to different CPUs. This bottleneck can become relevant if several dual-port 40G NICs are used. For 100G Ethernet, two dual-port NICs, attached to different CPUs of a system, can already overload this interconnect.

For 10G Ethernet, the only relevant bottleneck is the Ethernet bandwidth itself, the other interconnects of modern systems supporting PCIe 3.0, DDR4, and UPI exceed the bandwidth requirements of 10G. For 40G and 100G interconnect bandwidth, especially PCIe and CPU interconnect bandwidth may present a bottleneck. For a model predicting the bandwidth of

a 10G Ethernet system, the interconnects except for the Ethernet itself are irrelevant and do not need to be considered by such a prediction. However, for 40G and above, PCIe and CPU interconnect can become a factor in such a prediction.

4.1.2 CPU Bottleneck

Besides the interconnect bandwidth, the CPU time is another resource needed for software-driven packet processing. Rizzo [28] models processing costs from the perspective of a packet. He defines two components:

Per-byte costs: depend on the length of a packet, e.g., the costs for copying or encrypting a packet that increase linearly with packet length. Packet processing tasks that operate only on header data can have per-byte costs of 0.

Per-packet costs: are static costs occurring once for every processed packet. These are the dominating costs for typical processing tasks operating on packet headers, such as forwarding or routing. With per-packet costs dominating, a high packet rate becomes the most challenging scenario for packet processing applications. Therefore, worst-case scenarios are typically conducted at line rate with the minimal Ethernet packet size of 64 B.

We extend the view on CPU costs by taking the root cause of the CPU costs into consideration [39]. CPU time can be spent for two different reasons:

Data access costs: happen if the CPU waits for data to become available for processing. Data can only be processed if it is available in the CPU registers. The location of the data determines the actual costs, i.e., the higher a date is placed in the cache/memory hierarchy, the lower the access costs. Data access costs are essential for packet processing, as packet data must be loaded from the RAM or cache. Additionally, data structures such as routing tables must be available to perform packet processing tasks.

Calculation costs: are caused by the time spent on data manipulation. After data is loaded into the CPU registers, instructions are performed on the data. The complexity of the performed operations determines the calculation costs. The switching or routing of packets typically involves less complex arithmetic operations. Hence, it takes less time than the computationally complex encryption or decryption of packet data.

Modern CPUs incorporate independent units for fetching and processing data. Therefore, fetching and processing data can be performed in parallel, lowering the time either unit idles. However, this kind of parallelism has its limits as calculation and access costs are rarely evenly distributed for typical packet processing tasks. Access costs typically dominate for tasks such as routing or switching, calculation costs for encrypting or decrypting.

Both classifications can be combined, i.e., per-packet costs can be divided into access and calculation costs. The same holds for per-byte costs. As per-packet costs dominate the packet processing tasks investigated in this thesis, the described resource model focuses on per-packet rather than per-byte costs.

Location	Size	Access Latency [CPU cycles]
L1 Data	32 kB	4
L2 (unified)	256 kB	12
L3 (shared)	up to 8 MB	26-31

Table 4.2: Data Access Cost on Intel Sandy and Ivy Bridge CPUs (cf. Intel [46])

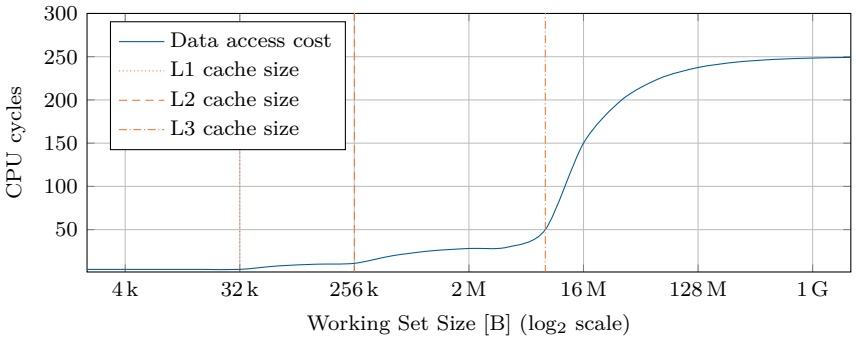


Figure 4.4: Ideal model for data access costs using random accesses on Ivy Bridge microarchitecture

4.1.3 Impact of Caching on Data Access Costs

Table 4.2 lists the cache sizes and data access costs for Intel CPUs based on the Sandy or Ivy Bridge microarchitectures, used for several measurements in this thesis. These access latencies demonstrate that caches have a significant impact on data access costs. The costs approximately rise by a factor of three when changing to the next higher level in the cache hierarchy.

The Sandy and Ivy Bridge microarchitectures offer separate instruction and data caches on L1; higher levels are unified, i.e., instructions and data share the same cache. For the analyses in this thesis, we focus on data caching abilities. L1 and L2 are non-shared caches. Hence, every physical CPU core has its own L1 and L2 caches. The L3 or LLC is shared across all available CPU cores. Through this shared cache, applications running on separate cores may still influence each other. The size of the LLC depends on the microarchitecture and the actual CPU model, the respective CPUs in this thesis have a typical LLC size of 8 MB. However, the LLC is an inclusive cache that includes all data contained in the L2 caches reducing the additional cache space the LLC effectively offers. [46]

If RAM is accessed, access costs increase further. The value of these costs depends on the type of RAM in use. The Sandy and Ivy Bridge CPUs support DDR3, we use DDR3-1333. For such a system, we measured approximately 250 CPU cycles for accessing data residing in RAM (cf. Section 5.1.5).

Figure 4.4 shows an idealized model for the average data access costs. This model assumes a data structure with different sizes that can use the cache and RAM exclusively. We do not consider the initial data access costs that happen if entries are accessed the first time and fetched into the cache. Besides, we assume that the data structure occupies the cache in the most efficient way possible, i.e., lower cache levels are preferred as long as space is available. The data accesses

happen at a random position in this data structure, so the CPU cannot predict the access pattern and prefetch data into higher cache levels before the access.

The access costs do not increase linearly but follow a stepwise pattern. Costs rise where the data structure exceeds the size of a cache level. The increase is not sharp; lower cache levels are still used but with a lower probability. Figure 4.4 shows that data access costs can rise by a factor of 50 when comparing L1 to RAM accesses.

4.1.4 Resource model

The resource model combines the relevant system resources to predict an upper bound for the system's packet processing performance—the interconnect bandwidth and computing resources. Our model uses all the individual limits of the available system interconnects for the calculation of the resulting upper bound for the interconnect bandwidth. For the measurements in this thesis, the per-packet costs are dominating. Therefore, we model the costs in CPU cycles per packet.

$$T_{max}^{interconnect} = \min(T_{max}^{Ethernet}, T_{max}^{PCIe}, T_{max}^{memory}, T_{max}^{QPI}) \tag{4.3}$$

$$T_{max}^{interconnect} = T_{max}^{Ethernet} \tag{4.4}$$

Equation 4.3 describes the maximum achievable throughput on a system. There, the minimum bandwidth of the involved components determines the overall bandwidth rendering all the other limits irrelevant for modeling. According to our previous analysis, the Ethernet bandwidth is typically the lowest and, therefore, the dominant one leading to the simplification in Equation 4.4.

$$T_{max}^{CPU} = \frac{f^{CPU}}{c} \cdot p \tag{4.5}$$

The function in Equation 4.5 describes the maximum CPU-bound throughput of a system. The computation capacity of the system is determined by the clock frequency f^{CPU} of the used CPU. The variable c represents the average per-packet costs of the packet processing task. These costs are measured in CPU cycles and contain access and calculation costs. Dividing the available frequency by the average per-packet costs determines the maximum number of packets the system can process per second. The packet rate can be converted to the throughput in Gbit/s by multiplying the packet rate with the average length of the packets. Packet size p includes the additional bytes between two packets sent on the physical layer, i.e., inter-packet gap, preamble, start of frame delimiter, and the frame check sequence.

Like the interconnect limits, the resulting overall throughput T_{max} is the minimum of the previous two Equations 4.4 and 4.5. This leads to Equation 4.6.

$$T_{max} = \min(T_{max}^{Ethernet}, T_{max}^{CPU}) \tag{4.6}$$

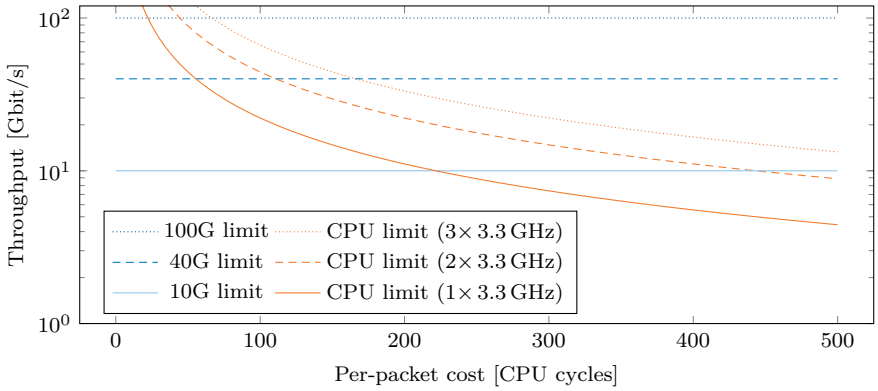


Figure 4.5: Model for packet processing with a packet size of 64 B (cf. Gallenmüller et al. [38])

A visualization of this resource model in Equation 4.6 is shown in Figure 4.5. There, the Ethernet bandwidth of the respective standard is given as a horizontal line. This Ethernet limit only considers the bandwidth independent of the packet size; if packet rate limits apply, additional lines must be introduced. If the Ethernet bottleneck limits the overall throughput of the system, we call the system bandwidth-bound. For low packet costs, the system is typically bandwidth-bound. If packet costs increase, the system becomes CPU-bound, meaning that the computing resources limit the system's overall throughput.

As long as no other interconnect limit is hit, the bandwidth-bound system performance, i.e., $T_{max}^{Ethernet}$, can be increased—either by upgrading the Ethernet standard or by adding NIC ports to the system. Figure 4.5 includes the three Ethernet standards for 10G, 40G, and 100G, demonstrating the effects of increased bandwidth. Additionally, three CPU configurations are visualized utilizing 1, 2, or 3 CPU cores. We chose a clock rate of 3.3 GHz. Adding CPU cores helps increase the system resources f, shifting the point at which the system becomes CPU-bound. Section 5.3 demonstrates that packet processing applications scale almost linearly with additional CPU cores. Another way to increase system performance T_{max}^{CPU} in a CPU-bound system is to lower the costs per packet c.

4.2 Key Results

This chapter introduces the resource model that predicts the throughput performance of packet processing systems. The resource model considers the two major bottlenecks that impact performance: the interconnect bandwidths and available CPU time. Exhausting either the bandwidth of one of the many system interconnects or the CPU processing capacity limits the overall throughput performance of packet processing, which can be predicted using the resource model.

The resource model can be implemented efficiently. We minimize the resource model's complexity by concentrating on the most relevant interconnects *(parsimony)*. Simultaneously, the simpler model, which considers only the system interconnect with the lowest bandwidth, is as expressive as a more complex model considering all system interconnects *(expressiveness)*. In

addition, we focus on parameters for the resource model that can be measured easily (*measurability*).

4.3 Author's Contributions

Section 4.1 is based on work by Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle [38], this publication itself is based on the Master's Thesis [39] by the author. Technical background about the system interconnects was added for this thesis and the analysis contains a new discussion for 40G and 100G Ethernet.

Chapter 5

Measuring and Modeling of High-Speed Packet Processing Systems

The introduction of 10G Ethernet led to the development of specialized frameworks to support the increased bandwidth. This chapter investigates the performance of these frameworks and applications based on them. We analyze a framework for building network functions (NFs) called Snabb, the high-performance software router MoonRoute, and an accelerated version of the intrusion prevention system Snort.

5.1 Comparison of Packet Processing Frameworks

Section 5.1 is based on joint work by Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle [38], this publication itself is based on the Master's Thesis [39] by the author.

Nowadays, 10G Ethernet adapters are commonly used in servers. However, due to overhead imposed by the network stacks' architectural design, the CPU quickly becomes the bottleneck, so that packet handling—even without any complex processing—is impossible at line speed for small packet sizes. Software frameworks for high-speed packet IO, e.g., netmap [28], DPDK [47], or PF_RING ZC [27], promise to fix this issue by offering a stripped-down alternative to the Linux network stack. Their performance increase allows using commodity hardware systems as routers and (virtual) switches [48], [49], network middleboxes like firewalls [50], or network monitoring systems [51]. Motivated by the potential gain, we analyzed the performance characteristics of these frameworks.

Chapter 4 introduced a modeling framework utilizing interconnect bandwidths and CPU resources to predict the performance of arbitrary packet processing applications. Here, we investigate how we can apply this modeling framework to programs based on the previously mentioned packet processing frameworks. Various measurements show the applicability of the resource model, with packet forwarding as the basic test scenario. Each measurement is designed to

investigate the influence of a specific factor on the forwarding throughput, e.g., the clock speed of the CPU, the number of processed packets per call, or the cache utilization. Moreover, the latency of packet forwarding is reviewed. All measurements are designed to ensure the comparability of our test results: running on the same CPU, equipped with the same 10G NICs while using packet forwarders applying the same algorithm for each of three frameworks, respectively, to provide a fair comparison between them.

Our comparison is organized as follows: In Section 5.1.1, we describe the state of the art in fast packet processing. Related work that serves as the basis for our research is presented in Section 5.1.2. Section 5.1.3 investigates potential bottlenecks for packet processing performance. In Section 5.1.4, we apply the resource model to high-performance packet processing frameworks. Subsequently, Section 5.1.5 presents our comparison of techniques for fast packet processing. We conclude with a summary of our results in Section 5.1.6.

5.1.1 Packet Processing in Software

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

Utilization of Hardware Features

On the hardware side, the performance increase, besides the higher bandwidth, came from offloading features that shifted the workload from the CPU to the NIC. Checksum offloading, for instance, relieves the CPU from CRC checksum handling. Now the NIC takes care of calculating the CRC checksum and adding it to the packet before transfer. On the receiving end, the NIC validates the checksum and drops the packet in case of an error, without involving the CPU. [20]

DMA allows the NIC to write or read packets directly into or from RAM bypassing the CPU. Modern NICs can even copy packets directly into the cache of the CPU, which leads to a further increase in performance [40], [52].

Another part of the speed-up comes from increased CPU performance. In addition to higher clock rates, the number of CPU cores has changed from one to a growing number of cores. NICs need to support multi-core architectures explicitly by distributing incoming packets of different traffic flows to different cores. One of these techniques is RSS, which allows scaling packet processing with the number of cores. [20]

Linux Network Stack

On the software side, the performance increase came, despite the support of the new hardware features, from a more efficient way to handle incoming traffic. The first approach of generating one interrupt per incoming packet was unsuitable for high packet rates due to livelocks caused by interrupt storms [53]. In such a livelock, the system is almost entirely occupied with handling the overhead caused by interrupts instead of processing packets.

NAPI, a network driver API introduced in the Linux kernel 2.4.20, reduces the number of interrupts generated by incoming traffic with the ability to switch to polling for packets during phases of high load, effectively reducing system overhead [53]. The NAPI-based network stack is sufficiently powerful to scale software routers to multiple Gbit/s [54], [55]. But even though performance improved, the Linux network stack primarily focuses on offering a full-featured general-purpose network stack for an OS rather than providing a high-performance interface needed for software router applications [56].

High-Speed Packet Processing

Compared with a general-purpose network stack like the one implemented in Linux, high-speed packet IO frameworks offer only basic IO functionality: Layer 3 and above must be implemented by the application, whereas the Linux network stack handles Layer 3 and Layer 4 protocols like IP and TCP. As a benefit, these frameworks offer increased performance compared with a full-blown network stack. In this thesis, we focus on the most important representatives netmap [28], PF_RING ZC [57], and DPDK [47]. All three frameworks require modified drivers and use the same techniques for acceleration:

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

netmap: netmap [28] exposes packet buffers to the application and uses standard system calls, like `poll()` or `ioctl()`, to initiate the data transfer. The work behind these system calls is reduced compared with a default network stack. These system calls only update the packet buffers and check the data provided by user programs for their validity to prevent crashes.

The network drivers of netmap are based on regular Linux drivers. As long as no netmap application is active, the driver works transparently for OS and traditional applications. Upon starting a netmap-enabled application, the NIC is put into a special *netmap mode*, i.e., the NIC becomes inactive for the OS and no packets are delivered to the standard OS interfaces and traditional applications. Instead, the packets are transferred to netmap-specific data structures where they are available to the netmap-enabled application. When closing this application, the driver switches back to transparent mode. Maintaining this compatibility in the driver allows for easy integration into a general-purpose OS. The FreeBSD kernel includes netmap as a means for high-performance packet processing [58].

Multiple applications have shown increased performance by adapting netmap: Click [59], a software router, the virtual switch VALE [48], and the FreeBSD firewall ipfw [50].

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

PF_RING ZC: PF_RING ZC does not use standard system calls but offers its own functions. The API of PF_RING ZC emphasizes convenient multi-core support [60]. It is NUMA aware, i.e., on systems with multiple CPU sockets, the packet buffers can be allocated in memory regions a CPU can directly access. Moreover, processes can be clustered for easy data sharing among them.

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

Ntop [61] offers several applications running on top of PF_RING ZC, for example, n2disk, a packet capturing tool, or nProbe, a tool for traffic monitoring.

DPDK: DPDK is a collection of libraries, which not only offer essential functions for sending and receiving packets but provide additional functionality like a longest prefix matching algorithm for the implementation of routing tables and efficient hash maps. DPDK relies on a custom userspace API similar to PF_RING ZC instead of traditional system calls used by netmap. The DPDK API [62] offers multi-core support, additional libraries used for packet processing, and features the highest degree of configurability among the investigated frameworks. The DPDK driver does not feature a transparent mode, i.e., as soon as this driver is loaded, the NIC becomes available to DPDK but is made unavailable to the Linux kernel regardless of whether any DPDK-enabled application is running or not. DPDK uses a special kind of driver aiming to do most of its processing in userspace. This UIO driver [63] still has a part of its code realized as kernel module, but its tasks are reduced. It only initializes the used PCI devices by mapping their memory regions into the userspace process.

A notable example of an application using DPDK, which gained attention, is an accelerated version of OvS [49]. An additional high-performance software switching solution is xDPd [64], which supports DPDK for network access. Click, a software router, also supports DPDK [59].

Other frameworks: The already mentioned frameworks are not the sole solutions offering high-speed packet processing capabilities in userspace.

PacketShader [65] is a packet processing framework using the general-purpose capabilities of GPUs for packet processing. It also features a separate engine for fast packet IO. The GPU part of PacketShader is not publicly available; only the code of the packet engine was released, which can be used on its own. The packet IO engine is currently not developed further, the repository was archived [66].

PFQ [67] is a framework that is optimized for fast packet capturing. It does not rely on specialized drivers like the other frameworks and can be used with every NIC as long as Linux supports this card. However, without modified drivers, the NIC cannot push the packets directly to user-space. The lack of this feature leads to a performance disadvantage when compared with the previously mentioned frameworks. A notable feature of PFQ is the integration of a Haskell-based domain-specific language for implementing packet processing algorithms [68]. PFQ focuses on providing a framework to make packet processing easy and safe rather than providing the highest possible performance. Therefore, the typical use cases for PFQ differ from the use cases of the other frameworks.

Snabb [69] is a packet processing framework with a focus on creating NFs. Similar to MoonGen, Snabb is optimized for flexibility and extensibility and relies on the scripting language Lua and the LuaJIT compiler for high performance. The framework provides a simple API to create user-defined NFs. A unique feature for Snabb is its NIC driver, which is entirely written in Lua. The Snabb framework, its driver, and the users' NFs are all written in this easy-to-learn scripting language. This common language helps users to understand and potentially modify the entire framework. Due to Snabb's focus on high-level NFs, it is not included in the detailed comparison of packet processing frameworks. A separate analysis of Snabb is given in Section 5.2.

5.1.2 Related Work

A survey of various packet IO frameworks was published by García-Dorado et al. [56]. The theoretical part of their investigation is comprehensive and the paper includes measurements showing selected aspects of these frameworks, e.g., the influence of the number of available cores and packet sizes on the throughput. They investigate the packet IO engine of PacketShader, PFQ, netmap, and PF_RING DNA, a predecessor of PF_RING ZC. DPDK and Snabb are not investigated. However, the authors only analyze packet capturing capabilities and neglect other aspects of packet processing.

Throughput measurements of software packet forwarding systems on commodity hardware have been conducted previously: Bolla and Bruschi [55] analyze a Linux software router. Dobrescu et al. [54] published studies of software router performance and the influence of various workloads. The highest throughput of a software solution implementing an OpenFlow switch with DPDK was presented and measured in [70]. We also measured the throughput of Linux-based forwarding tools in previous work [71]. These measurements allow a direct comparison with results from this thesis because they were performed on the same test system.

The latency of a Linux software router was also measured by Bolla and Bruschi [55]. Angrisani et al. [72] describe a technique to measure different parts of packet processing systems using commodity hardware based on internal queuing. Rotsos et al. [73] present an FPGA-based

method to measure the latency of various software and hardware OpenFlow switches. They present measurements for OvS running on Linux as an example. A discussion of latency in software routers can also be found in [74]. The authors describe a method that can be used to distinguish the latency introduced by queuing from the processing delay.

The selected literature shows that the performance of the Linux networking part is thoroughly researched and well-known. There are also papers investigating a specific framework exclusively. However, measurements require similar test conditions, i.e., comparable hardware and software setups, to ensure comparability. The paper by García-Dorado et al. [56] provides those conditions but measures only a few selected aspects. Therefore, we try to give a fair comparison by testing each framework on the same hardware. We include additional measurements, such as the transmission of packets or latency determination. We also introduce a novel model to provide a basic understanding how packet processing applications work and how their performance can be estimated.

5.1.3 Performance Considerations

We present a model that provides insights into the performance of the packet processing applications built for high-speed IO frameworks. It uses the main factors influencing performance to provide an upper bound for the capabilities of a software-based packet processing system that were analyzed and modeled in Section 4.1. In this section, we build upon this modeling framework and derive a model to describe high-performance packet processing frameworks.

Upper Bound for Packet Processing

The investigated system uses 10G Ethernet attached via an 8× PCIe 2.0 port, a single CPU with a clock frequency of 3.3 GHz, and dual-channel DDR3-1333 memory. With 10G Ethernet offering the lowest bandwidth of the available interconnects, the entire system can become bandwidth-bound at a throughput of 10 Gbit/s. The system becomes CPU-bound in case the computing resources of the CPU are exhausted. According to Equation 4.6, the throughput of the overall system (T_{max}) is the minimum of both upper bounds—bandwidth ($T_{max}^{Ethernet}$) and CPU (T_{max}^{CPU}).

Equation 4.5 defines a function to calculate T_{max}^{CPU} based on the clock frequency of the CPU f^{CPU}, the average per-packet costs c, and the packet size p. In the following, we want to analyze and model the consumption of CPU resources for high-performance packet processing frameworks and applications based on them. The value for T_{max}^{CPU} depends on the resources provided by the CPU—the cycles. These processing cycles can either be used to handle packets or to process other tasks. Costs for an individual packet are represented by c_i^{packet}. All costs for other processing tasks running on the CPU are summed up in c^{other}. To successfully execute a task with the given costs, these costs may not exceed the available computing resources f^{CPU}, which leads to the Inequation 5.1.

$$f^{CPU} \geq c^{other} + \sum_{i=0}^{n} c_i^{packet} \tag{5.1}$$

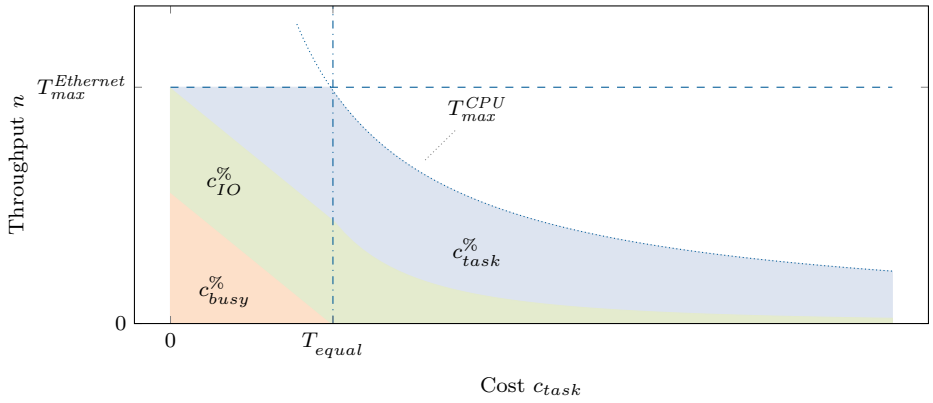


Figure 5.1: Model for packet processing (cf. Gallenmüller et al. [38])

The total costs of packet processing in CPU cycles are determined by the right side of Inequation 5.1. This sum contains the number of processed packets represented by the number of packets per second n and the individual costs c_i^{packet} of the packets. The maximum number of cycles per second f^{CPU} is a fixed value depending on the hardware. To get the maximum packet rate the system can handle, n has to be maximized with respect to Inequation 5.1.

Figure 5.1 shows the combination of the two upper bounds T_{max} as combination of $T_{max}^{Ethernet}$ and T_{max}^{CPU} with respect to growing costs per packet described by the x-axis. For this section, only the dashed and dotted lines are relevant. As long as $T_{max}^{Ethernet}$ is reached, the costs T_{max}^{CPU} are low enough to be entirely handled by the available CPU, the traffic is bound by the limit of the NIC. At point T_{equal}, the throughput begins to decline. Beyond this point, the CPU processing time does not suffice the traffic capabilities of the NIC, i.e., the traffic becomes CPU-bound and the throughput subsequently sinks.

The costs per packet determine how many packets can be processed without surpassing the computational limit T_{max}^{CPU}. The actual shape of T_{max}^{CPU} cannot be determined as it depends on the traffic and the processing task. Regardless of the precise shape of this curve, the outcome stays the same, i.e., higher per-packet costs decrease the throughput. The hyperbolic shape of T_{max}^{CPU} depicted in Figure 5.1 holds for packet processing frameworks and is explained in detail in the following section.

5.1.4 High-Performance Prediction Model

According to Rizzo, packet processing costs can be divided into per-byte and per-packet costs, with the latter dominating for IO frameworks; i.e., it is only slightly more expensive to send a 1.5-kilobyte packet than sending a 64-byte packet [28]. This leads to two assumptions to be made. The first assumption is that the per-packet costs are constant for high-performance IO frameworks. The second one is that experiments are performed under the most demanding circumstances if the highest packet rate is chosen, i.e., 64-byte packets have to be used.

In case of constant costs per packet $\forall i \ : \ c_i^{packet} = c_{const}^{packet}$ and a dedicated core for packet processing leads to $c^{other} = 0$. If the packet processing application itself also generates a

constant load per packet and the high-performance frameworks have roughly constant costs per packet and use dedicated cores for packet processing, Inequation 5.1 can be simplified to the following inequation:

$$f^{CPU} \geq n \cdot c_{const}^{packet} \tag{5.2}$$

If packet processing includes actions that depend on the type of packet or the traffic characteristics, the computation may become infeasible. Such a scenario may be packet monitoring where certain types of packets require additional CPU cycles for further analysis [75]. Without restriction to specific traffic patterns, it is still possible to approximate the overall costs with average per-packet costs or to do a worst-case estimation.

Due to the architecture of the frameworks, which all poll the NIC in a busy waiting manner, an application uses all the available CPU cycles all the time. If the limit of the NIC is reached, but $n \cdot c_{const}^{packet}$ is lower than the available CPU cycles, the cycles are spent waiting for new packets in the busy-wait loop. If these costs are included, a new value is introduced $c_{const}^{*packet}$ and both sides of the former Inequation 5.2 are now balanced:

$$f^{CPU} = n \cdot c_{const}^{*packet} \tag{5.3}$$

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

$$c_{const}^{*packet} = c_{IO} + c_{task} + c_{busy} \tag{5.4}$$

1. c_{IO}: These costs are used by the framework for sending and receiving a packet. The framework determines the amount of these costs. In addition, these costs are constant per packet due to the design of the frameworks by completely avoiding operations depending on the length of the packet, e.g., buffer allocation.
2. c_{task}: The application running on top of the framework determines those costs, which depend on the complexity of the processing task.
3. c_{busy}: These costs are introduced by the busy waiting on sending or receiving packets. If throughput is lower than T_{max}, i.e., the throughput becomes CPU-bound, c_{busy} becomes 0. The cycles spent on c_{busy} are effectively wasted as no actual processing is done.

Combining Equations 5.3 and 5.4 leads to:

$$f^{CPU} = n \cdot (c_{IO} + c_{task} + c_{busy}) \tag{5.5}$$

Figure 5.1 depicts the behavior of the throughput while gradually increasing c_{task} as described by Equation 5.5. The highlighted areas show the relative part of the three components of $c_{const}^{*packet}$. Each area depicts the accumulated per-packet costs of their respective component x called $c_x^{\%}$.

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

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

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

5.1.5 Performance Comparison

The available CPU cycles are the main limiting factor of software packet processing. Subsequently, the throughput of a packet processing application heavily depends on the number of CPU cycles available for its processing task. This number of CPU cycles is influenced by multiple factors and the following measurements present a selection of factors we consider relevant for real-world applications: The overhead caused by the complexity of packet processing, the time the CPU spends waiting for data to arrive in the CPU cache, and the effect of different batch sizes, i.e., whether the packet throughput rises if more packets are processed per call. For every factor, a dedicated measurement is performed. We investigate the batch size, as it, in particular, determines the queuing delay of the packets on the processing system and latency during packet forwarding.

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

Measurement Setup

Instead of the typical two-server setup, this setup uses three servers with the load generator split into a load generator and a traffic sink. The split has been necessary due to hardware restrictions: only single port NICs were available, and the traffic source and sink could only hold a single NIC. The DuT is configured as a forwarder running the investigated frameworks, connected to the load generator and the traffic sink via 10G links. The forwarder is equipped with a dual-port Intel X520-SR2 NIC, the load generator and sink use single port X520-SR1 NICs. These cards use PCIe v2.0 with 8 lanes, which offers a usable link bandwidth of 32 Gbit/s in both directions. The Intel cards were chosen, as driver implementations exist for each of the investigated frameworks. This avoids a possible performance impact introduced by different kinds of NICs. The server acting as forwarder runs on an Intel Xeon E3-1230 V2 CPU. The clock speed was fixed to 3.3 GHz, with power conserving mechanisms, Turbo Boost, and Hyper-Threading deactivated to make the measurements consistent and repeatable.

The forwarder statically forwards packets between the two interfaces without consulting a routing or flow table. It modifies a single byte in the packet to ensure that the packet is loaded into the Level 1 cache. Forwarding is done in a single thread pinned to a specific core.

As performance depends on the number of processed packets rather than the length of the individual packets, we use CBR traffic with the minimum packet size of 64 B for all measurements in Section 5.1 to maximize the load on the frameworks. The packets are counted on the sink using the statistics registers of the NIC.

We conducted measurements with a version of netmap, which was published on March 23, 2014, in the official repository [25], PF_RING ZC version 6.0.2 [27], and DPDK version 1.6.0 [47].

Our packet generator MoonGen [16] was used for latency measurements. It uses hardware features of our Intel NICs for sub-microsecond latency determination.

Every data point in our performance measurements is an average value. This value is calculated from 30 single measurements over a period of 30 s. Confidence intervals are omitted as results are stable and repeatable for all frameworks. An observation also made by Rizzo in the initial presentation of netmap [28].

Determine the Transmission Efficiency

All of the frameworks can forward packets at full line rate with a single CPU core for the tested hardware. To measure the transmission efficiency expressed by the CPU load caused by packet transmission, the CPU load generated by each framework needs to be compared. In Equation 5.5 this efficiency is referred to as c_{IO}. A low number of cycles spent on c_{IO} increases the number of cycles available for the actual packet processing task, i.e., c_{task}. In return, this allows more demanding applications to be built using more efficient frameworks without performance penalties.

Known approaches for measuring CPU load: Due to their architecture (cf. Section 5.1.1), excessive polling on the NIC causes the CPU cores used by the frameworks to be under full load at all times. Therefore, a simple comparison of CPU usage by a tool like `top` does not work. For this kind of measurement, there is no way to tell the relative portions of the three components of $c_{const}^{*packet}$ in Equation 5.5 apart.

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

Rizzo measured efficiency by reducing the CPU clock frequency until the throughput of the NIC was beginning to decline [28]. At this point, no busy-wait cycles happen, as depicted in Figure 5.1. This results in a c_{busy} value of 0. The packet processing task was simplified so that this component named c_{task} can also be neglected. Only the component c_{IO} remains, which is the efficiency of the framework. However, even at the lowest supported clock speed (1.6 GHz) in

our test setup, the forwarders transmitted at line rate. Therefore, this solution could also not be applied.

Novel method: To overcome the flaws of the previously presented methods for determining efficiency, we introduce a novel method for our measurements. Therefore, we add a piece of software, producing a constant load per packet on the CPU. The load can be specified as a number of CPU cycles to wait. This value can be increased until the throughput begins to decline. Intel provides a benchmark method [76] based on a clock counter called time stamp counter (TSC). We used this guide to design and calibrate this load mechanism. The code containing the load generation and benchmarking mechanisms is publicly available [77].

At the point of decline, c_{busy} is known to be 0, c_{task} is known by design. Subsequently, c_{IO} can be calculated. For this experiment, the forwarders were modified to implement this emulated CPU load c_{task} by spending a predefined number of CPU cycles per packet beside the framework's packet IO operations. The forwarders do not perform any lookup operation. Hence, the basic performance tests ignore cache effects that impact more complex packet processing applications that require lookups in a data structure (e.g., a forwarding or flow table). Therefore, we can assume that the basic forwarding applications spend a fixed number of CPU cycles per packet.

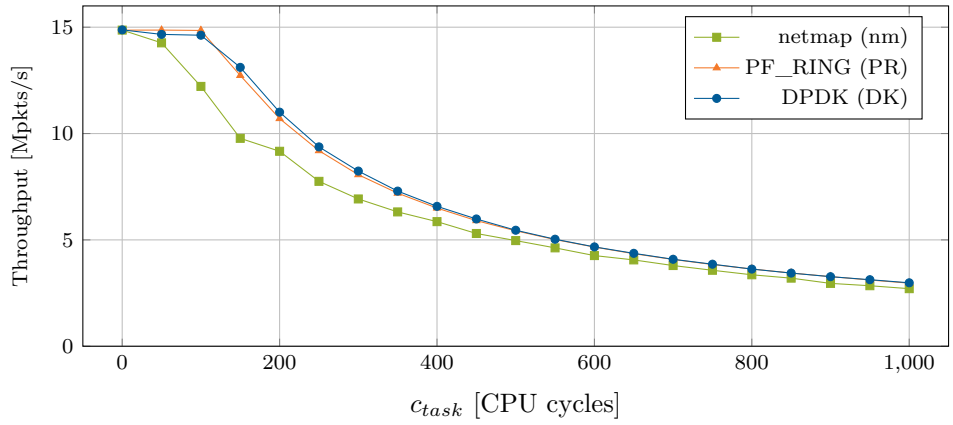
Measure the Transmission Efficiency

Figure 5.2a presents the results of throughput measurements with different CPU loads for the task emulator. As anticipated by our model in Figure 5.1, an increasing workload decreases the measured throughput. In the next step, we get back to our goal of measuring the per-packet CPU load consumed by each framework. To forward a packet, a CPU core dedicates cycles for transmission (c_{IO}), i.e., for receiving and sending a packet, cycles for the emulated task (c_{task}), and possibly cycles to poll the NIC unnecessarily (c_{busy}).

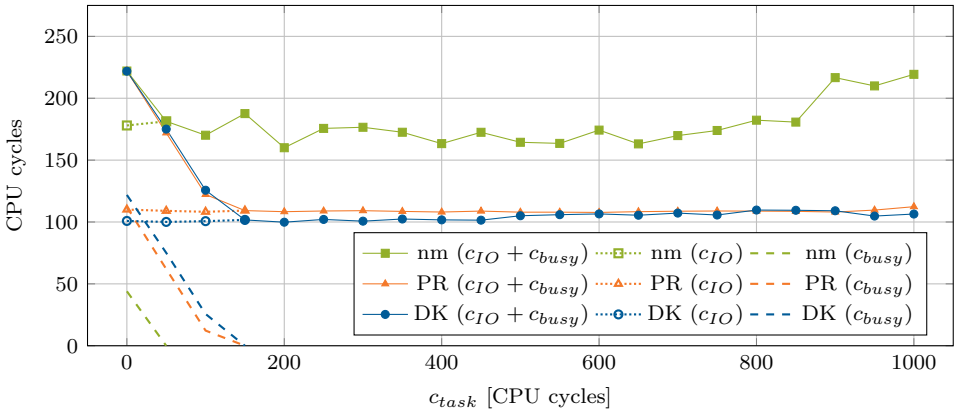
Knowing f^{CPU}, and taking $T_{max}^{Ethernet}$ and T_{max}^{CPU} from Figure 5.2a allows for the calculation of $c_{busy}+c_{IO}$ by applying Equation 5.5. These results are shown in Figure 5.2b for each framework. Starting at around 220 cycles for $c_{busy}+c_{IO}$ the graph decreases until the throughput is no longer limited by the 10 Gbit/s line rate. At this point, the throughput becomes limited by the CPU and no busy-wait cycles happen any longer, i.e., $c_{busy} = 0$. This allows for the separation of the two components, c_{busy} and c_{IO}, into two individual graphs, also depicted in Figure 5.2b.

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

We measured a c_{IO} of approximately 900 cycles for forwarding applications based on the Linux network stack in previous work [71]. This means that the frameworks discussed in this thesis can lead to a nine-fold performance increase over traditional network applications.



(a) Forwarding with an emulated task



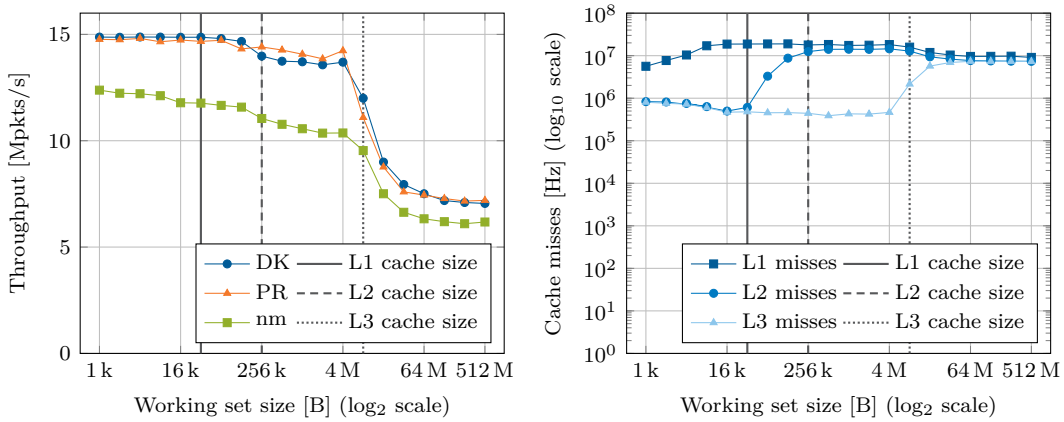
(b) Transmission cycles c_{IO} & busy polling cycles c_{busy}

Figure 5.2: Transmission efficiency measurements (cf. Gallenmüller et al. [38])

Influence of Caches

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

The necessary time to access data residing in RAM is shortened by the ability of modern CPUs to buffer accesses to RAM by integrating a hierarchy of several caches differing in size and access time. To test for different scenarios with only partly filled caches, the size of the data structure was made adaptable. The software influences what is put into cache indirectly by accessing data in RAM, which is then put into the cache or by giving hints to memory addresses via specialized commands. To optimize for typical access patterns, data close to already accessed addresses can be prefetched by the CPU before it is accessed [46]. Our tests showed that if a data structure is accessed linearly, this prefetching is working efficiently enough to hide the slow access speed to RAM. In the scenario of a routing table, the data to be accessed is determined by the traffic and the access pattern is likely to be non-linear.



(a) Performance with memory accesses

(b) Cache misses for DPDK

Figure 5.3: Cache measurements (cf. Gallenmüller et al. [38])

To mimic a worst-case scenario, the addresses accessed were randomized. Aiming for a realistic scenario, the prefetching was counteracted using a circular linked list with a random access pattern. This was achieved by randomly chosen links between the list elements while ensuring that the permutation contains a single cycle so that all memory locations are accessed once when the entire list is traversed. This guarantees random access on RAM or cache by iterating one step through the list for each received packet. The size of the linked list can be varied to emulate different routing or flow table sizes. An implementation of this data structure is publicly available [77].

Figure 5.3a depicts the throughput of the investigated frameworks in relation to the list size of our task simulator. For every packet processed, one emulated table lookup was performed. To investigate CPU-limited, rather than NIC-limited, throughput, a constant CPU load of 100 cycles was introduced, the point in Figure 5.2a where the throughput was beginning to decline for all three frameworks. This offset explains the lower throughput of netmap in Figure 5.3a, as expected from the data in Figure 5.2a.

The CPU in our test server has three cache levels, L1, L2, and L3 with 32 kB, 256 kB, and 8 MB, respectively [46]. Measurements showed that the average access time is 10 cycles for list sizes $\leq$ 32 kB, grows to 20 cycles for list sizes $\leq$ 256 kB, increases to 60 cycles for list sizes $\leq$ 8 MB, and finally reaches 250 cycles on average for list sizes larger than that. The measured access times are higher than specified by the manufacturer (cf. Table 4.2). Our measured costs include the processing costs for the data structure. As the L3 cache is shared across all cores, these access times include the additional accesses by other running processes.

The graph in Figure 5.3a shows no clear transition from L1 to L2 due to the low 10 cycle increase. The decline at around 256 kB is visible due to the higher speed difference between L2 cache and L3 cache. The next drop in the graph is the transition between L3 cache and non-cached RAM accesses.

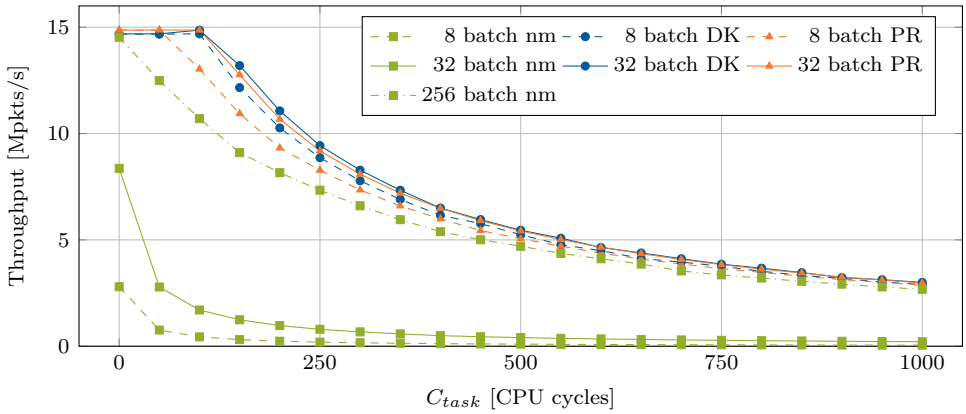


Figure 5.4: Throughput influenced by batch sizes (cf. Gallenmüller et al. [38])

DPDK is slightly slower than PF_RING ZC when the data structure fully occupies the L2 cache. This means that DPDK has a higher cache footprint than PF_RING ZC.

Figure 5.3b plots the cache misses, obtained by reading the CPU's performance registers. Only the results for DPDK are given. The results for netmap and PF_RING ZC are similar and excluded from the graph to improve readability. The number of cache misses starts at a certain level and begins to rise as a cache fills until the size of the test data exceeds the respective cache size. This observation holds for every cache level.

The data of Figure 5.3a can be used to test our model against a different problem. In contrast to the previously fixed load per packet, in this experiment, the load per packet was determined by the cache access times. However, even under these circumstances, the model provides a good estimation if average per-packet costs are used. At a list size of 256 MB, the average costs to access a list element, c_{task}, are 250 cycles. Taking the 100 extra cycles into account, leads to average costs of 350 cycles for c_{task}. For DPDK, the c_{IO} is roughly 100 cycles and f^{CPU} is 3.3 GHz. The expected throughput is 7.3 Mpkts/s with our model and the measured value in Figure 5.3a is 7.1 Mpkts/s. The minor difference can be explained by the fact that the test data structure also competes for cache space with data required by the framework, which results in additional overhead beyond the cache miss when sending or receiving packets. Therefore, the size of the data structures required for routing also needs to be considered when designing a software router.

Influence of Batch Sizes

In the following measurements, we analyze the influence of the batch size, i.e., the number of packets handled by one API call. The tests shown in Figure 5.4 were conducted using different batch sizes with increasing CPU load using the task emulator. For each iteration of the test, the batch size was doubled, starting at a batch size of 8 up to a batch size of 256. The results show that each framework profits from larger batch sizes. PF_RING and DPDK reach their highest throughput at a batch size of 32. Therefore, the larger batch sizes are omitted for those frameworks in Figure 5.4 because they also do not have adverse effects on the throughput.

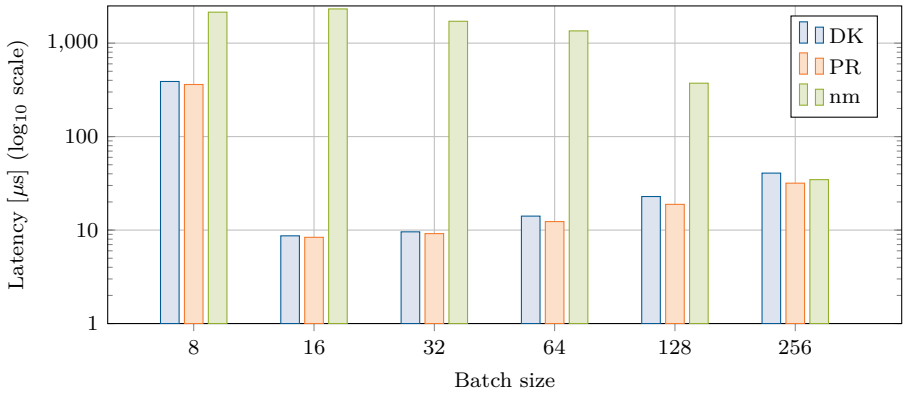


Figure 5.5: Average latency by batch size (cf. Gallenmüller et al. [38])

netmap needs a batch size of at least 256 to reach a throughput performance close to the other two frameworks. This is due to the relatively expensive system calls required to send or receive a batch (cf. Section 5.1.1).

Latency

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

We used the IEEE 1588 hardware timestamping features of the Intel 82599 controller to measure the latency of the forwarding applications [20]. The packets are timestamped in hardware on the source and sink immediately before sending and after receiving them from the physical layer. The timestamps do not include any software latency or queuing delays on the source and sink. This achieves sub-microsecond accuracy. [16]

Figure 5.5 shows the average latency for different batch sizes under a packet rate of 99 % of the line rate and no additional workload. Using line rate with CBR traffic causes delays after a minor interruption (like printing statistics) because it is not possible to send faster than the incoming traffic. The latencies were acquired by sending timestamped packets periodically (up to 350 pkts/s) at randomized intervals using a different transmit queue on the load generator. The timestamped packets are indistinguishable from the regular load packets for the forwarding application.

Both DPDK and PF_RING ZC are overloaded with a batch size of 8, netmap with all batch sizes smaller than 256, as described in the previous section. This causes all queues to fill up and the applications exhibit a worst-case behavior that is typical for a system that is overloaded. DPDK and PF_RING achieve an average latency of 9 μs with a batch size of 16 and the latency then gradually increases with the batch size. PF_RING ZC gets slightly faster than DPDK for larger batch sizes. netmap achieves an average forwarding latency of 34 μs with a batch size of 256.

These latencies can be compared with other forwarding methods and hardware switches: Rotsos et al. [73] measured a latency of 35 μs for OvS under light load and 3 μs to 6 μs for hardware-based switches. Bolla and Bruschi [55] measured 15 μs to 80 μs for the Linux router in various scenarios without packet loss and latencies in the order of 1000 μs for overload scenarios.

5.1.6 Conclusion

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

We described the processing performance of high-speed packet IO frameworks. Starting with a model describing packet processing software in general, this model is gradually adapted to reflect applications using high-performance frameworks. For our experiments, we rely on a precisely generated load on the CPU. We release the tools that we used for load generation as a separate library called SHEEP [77]. SHEEP consists of two tools—CPULoader and CacheLoader—that can be used independently.

Our experiments with these tools showed the performance characteristics predicted by our resource model. Thus proving the assumptions right, which we made during the development of this model. The CPU time spent on receiving and transmitting packets, for instance, remained constant despite the influence of the varying processing times per packet. Further measurements showed that this model could be applied to estimate processing tasks, which can be approximated with a constant average load. A possible use case for this model is to evaluate the PC systems' eligibility for specific packet processing tasks.

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

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

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

5.2 High-Speed Packet Processing for Network Function Chaining

Section 5.2 is based on collaborative work between Wolfgang Hahn, Borislava Gajic, Florian Wohlfart, Daniel Raumer, Paul Emmerich, Sebastian Gallenmüller, and Georg Carle [78].

The availability of high-speed packet processing frameworks allows the creation of powerful packet processing applications. In the following, we present a methodology for designing and measuring such applications.

Network function chains (NFCs) are a concept first described by the European Telecommunications Standards Institute (ETSI) in 2012 [79]. This concept describes the functionality, formerly running on dedicated hardware devices, shifting into software running on off-the-shelf servers. To create complex applications, basic packet processing tasks, so-called network functions (NFs), are combined to form an NFC. Snabb [69] is a packet processing framework fostering such an architecture, similar to the frameworks presented in Section 5.1.

Overprovisioning of resources harms efficiency and the lack thereof impacts application performance; two effects equally unwelcome when it comes to the design and operation of network applications. Knowing the performance of the available architecture would allow the efficient operation of NFs [79]. A straightforward way to determine the performance of an NFC is the measurement of the NFC under different load and configuration scenarios. Creating and setting up such a realistic measurement environment can be extensive and complex. Section 5.1 demonstrates that the CPU is the main bottleneck for all packet processing applications except for simple L2 forwarders. Subsequently, the performance of an NF depends on the CPU load caused by the application. By emulating the CPU load caused by different NFs, the performance can be measured without the need to create the actual NF or a complex configuration.

For emulating the CPU load, we use SHEEP [77]. Like MoonGen, Snabb uses LuaJIT's FFI to include and call C libraries such as SHEEP conveniently. That feature allowed us to create an NF implementing SHEEP. The SHEEP-enabled NF can create a static processing load by waiting for a specified number of CPU cycles for every received packet. This can be used to create packet processing tasks causing a constant overhead, e.g., the encryption or decryption of a fixed number of bytes. Besides constant overhead, dynamic costs can occur, e.g., the prefix lookup during routing or a rule lookup for a firewall NF. Dynamic costs are influenced by the size of the CPU caches and lookup data structures or the number of lookups per packet. To emulate the CPU load of an NF, the SHEEP-enabled NF can be configured with three parameters: a fixed delay of CPU cycles per packet, the number of data structure accesses in the cache loader data structure per packet, and the size of the data structure. Static and dynamic load can be used exclusively or combined. Following that approach, the performance of the entire NFC under specified CPU load conditions can be emulated.

We adapt the resource model to NFC in Section 5.2.1 before validating our assumptions with measurements in Section 5.2.2. We conclude our findings in Section 5.2.3.

5.2.1 Network Function Chain Model

An NFC is a packet processing application; therefore, we can use the model introduced in Section 5.1.3 to predict its performance. The original model, given in Equation 5.5, considers three cost components c_{IO}, c_{task}, and c_{busy}. Snabb receives and sends packets (c_{IO}), processes packets (c_{task}), and applies a busy-polling reception (c_{busy}), enabling an application of our model to this use case.

A characteristic element of NFCs is their chaining functionality. Chaining costs occur when handing over packets from one NF to another NF. Therefore, we introduce a new cost component into our model called c_{chain}. The costs for chaining depend on the number of chain elements e and the costs for chaining two elements $c_{chain-element}$. The initial reception and the final transfer of a packet are not part of the chaining costs but part of c_{IO}. Therefore, an NFC with a single NF has chaining costs of 0. The per-packet chaining costs can be calculated using Equation 5.6.

$$c_{chain} = (e - 1) \cdot c_{chain-element} \tag{5.6}$$

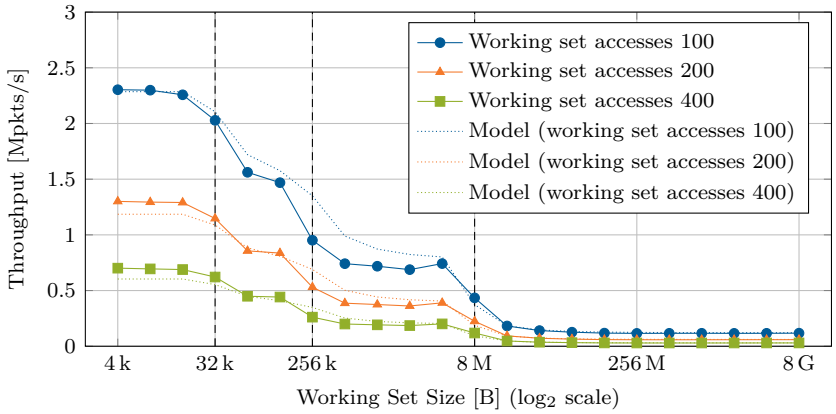
CPU costs of an application (c_{task}) can be divided into two components: the calculation and data access costs (cf. Section 4.1.2). For complex packet processing, a typical task for NFCs, we consider both components separately to provide a deeper understanding of packet processing performance and potential bottlenecks. Therefore, we split c_{task} into two components $c_{task-calc}$ and $c_{task-access}$, for calculation and access costs. Utilizing SHEEP, we can emulate both components separately. Applying these extensions to the original model leads to Equation 5.7.

$$f^{CPU} = n \cdot (c_{IO} + c_{chain} + c_{task-calc} + c_{task-access} + c_{busy}) \tag{5.7}$$

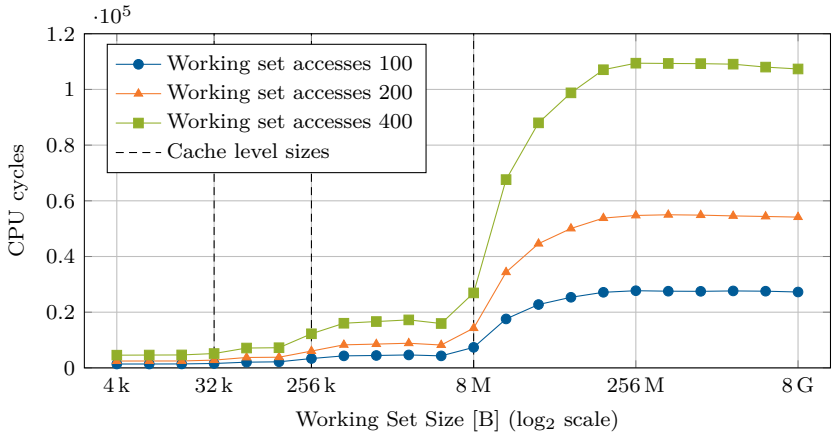
Equation 5.5 can be used to predict the upper bound for throughput performance of an NFC. The model uses the CPU resources, its frequency f^{CPU}, that limits the number of processed packets n depending on the per-packet costs c_*.

5.2.2 Network Function Chain Measurement

To check the prediction of Equation 5.7, we conduct measurements. For these measurements, we used the typical two-server setup consisting of a load generator and a DuT. The DuT sever was equipped with an Intel Xeon E3-1230 CPU (4 cores, 3.2 GHz), 16 GB RAM, and an Intel 10G X540 NIC. The server was running Debian Linux (Jessie, kernel version 3.16) and Snabb in version 2016.11. The DuT runs Snabb using different NFCs forwarding packets from an ingress to an egress port. The entire chain uses a single CPU core. The load generator used MoonGen to generate 64-byte UDP packets in a CBR pattern.



(a) Throughput measured and modeled



(b) Average measured per-packet costs (cf. Hahn et al. [78])

Figure 5.6: Forwarding with SHEEP-enabled NF

To determine the IO costs for Snabb, we use a SHEEP-enabled NF. We configure per-packet calculation costs ($c_{task-calc}$) of 500 cycles per packet, to avoid spending cycles for busy polling, i.e., c_{busy} is 0. Only one chain element is configured, so c_{chain} is 0, $c_{task-access}$ is configured to 0 in the NF. Using this measurement, we calculate per-packet IO costs, c_{IO}, of approximately 100 cycles.

Chaining costs can be determined using the identical forwarding NF five times in a row. This longer chain increases the forwarding costs and the 10G line rate could no longer be reached, i.e., c_{busy} is 0. Knowing the c_{IO} costs, and setting c_{task-*} to 0, allows us to calculate c_{chain}. Applying Equation 5.6 to our results leads to chaining costs of 35 cycles ($c_{chain-element}$) between two NFs.

With known chaining and IO costs and configurable costs for the processing task, we can try to predict the per-packet costs of an NFC using Equation 5.7. To model the data access costs, $c_{task-access}$, we used the ideal model presented in Section 4.1.3. To create more realistic approximations for $c_{task-access}$, we add cycles to reflect the overhead of SHEEP. Measurements

have shown an overhead of 10 cycles per access. We further reduce the cache size that can effectively be used for storing the data structure, as parts of the caches on L2 or L3 are used as instruction cache. Measurements have shown that a value of 85 % is realistic for the investigated application.

Figure 5.6b shows the per-packet data access costs for different configurations of the data set size and data accesses. We choose data structure sizes between 4 kB and 8 GB and configure the number of data structure accesses per packet to 100, 200, or 400. The shape of this curve presents the same step-wise pattern as our ideal model of data accesses (cf. Figure 4.4). Figure 5.6a shows modeled and measured data for the average throughput.

The modeled data presents a rough estimate of the actual measured data. The model has a low deviation for RAM accesses. However, the cache accesses show a higher deviation from the measurement. One reason for that behavior is the higher impact of small deviations for cache accesses, i.e., a small error in the prediction of cache access costs can lead to a high impact on the measured throughput. Another reason for the overestimation in the area of the LLC is the shared nature of this cache level. Other running tasks that use the LLC are not considered by our data access cost model.

Varying the data set size leads to non-linear behavior, i.e., steep increases of the per-packet costs are visible at the cache level limits. We observe the most significant difference when the working set size exceeds the L3 cache. For the number of data structure accesses, a linear trend is visible—doubling the number of accesses leads to a two-fold increase in per-packet costs.

5.2.3 Conclusion

We demonstrate a successful application of the resource modeling framework to NFC. To meet the requirements of NFC, we introduce chaining costs as a new component of the resource model. Our measurements show that Snabb offers IO costs of 100 cycles—the same IO costs per packet as DPDK. Further, we demonstrate that our model can provide a rough estimation of the throughput performance of NFCs. The modeling of the access costs is less robust than calculation costs, due to the dynamic behavior of caches themselves and the shared LLC that can be influenced by the OS or other applications.

5.3 High-Performance Software Router

Section 5.3 is based on collaborative work between Sebastian Gallenmüller, Paul Emmerich, Rainer Schönberger, Daniel Raumer, and Georg Carle [80] and a technical report by Paul Emmerich, Sebastian Gallenmüller, Rainer Schönberger, Daniel Raumer, and Georg Carle [81].

Creating quick and dirty prototypes is a simple and effective way to demonstrate the feasibility of new ideas in network research. Though, a small-scale proof-of-concept may lack the performance needed to apply them to real-world test cases. Thanks to powerful packet processing frameworks such as netmap and DPDK, high-performance packet forwarding systems can be implemented in software today.

We present MoonRoute, a framework dedicated to developing powerful software routers. It is built on top of DPDK and utilizes a highly parallelized architecture to achieve high performance (see Section 5.3.1). MoonRoute offers methods to reuse existing libraries and a scripting interface for easy extensibility (see Section 5.3.2). We evaluate an example implementation based on the MoonRoute framework, demonstrate and model its performance, and compare it with other relevant software routers (see Section 5.3.3). We end our description of MoonRoute with a short summary (see Section 5.3.4).

5.3.1 High-Performance Design

MoonRoute leverages hardware features of modern NICs, such as RSS, to distribute packets according to specified parameters across multiple CPU cores. This functionality for almost perfect scaling across CPU cores forms the basis to implement scalable multi-core packet processing applications. To profit from such a hardware design, the software must be programmed accordingly. The threads must be able to run as independently as possible to not interfere with each other.

The underlying architecture of MoonRoute consists of two different kinds of threads or packet processing components—the *fast path* and the *slow path*. The fast path is concerned with processing many packets requiring rather simple processing, i.e., reaching a routing decision and forwarding packets—fast paths are simple and fast. Other more complex operations, such as generating an ICMP response for timed-out IP packets, are handled by the slow path. These operations require a more complex control flow, but occur less often than the simple packets—slow paths are versatile and slow. Building a router exclusively from slow path components would be possible; however, many functions would only be used rarely. To achieve high performance, our reference implementation uses both types of components: Several fast path components are distributed to different CPU cores and a single slow path component running on a separate core. All components utilize lock-free queues, avoid shared data structures where possible, and employ read-only data structures where information sharing between different threads is necessary. This optimized multi-thread design transfers the multi-core scalability provided by hardware into software.

A widely-used method to increase the performance of software packet processing systems is batching. In MoonRoute, functions called by fast path components accept, process, and return packets in batches to maximize throughput. Modules can exclude packets from further processing by building new batches (*rebatching*) or keeping the batches but flagging the packets for exclusion from further processing by subsequent functions (*flagging*). Both methods show disadvantages in certain situations: rebatching introduces additional overhead for building new batches and flagging leads to inefficiencies for large batches containing only a few packets to process. Our router implementation uses both methods to optimize performance. Flagging is used in between function calls in the fast path. Packets are rarely excluded from routing; therefore, flagging avoids the overhead for rebatching. Flagged packets need to be handled by the slow path. As the slow path usually handles only a few packets, sending whole batches with few flags introduces unnecessary load on the slow path. In this situation, the rebatching approach would slow down the fast path, whereas the flagging approach would decrease the slow

path performance. For this situation, MoonRoute introduces a novel hybrid approach between flagging and rebatching we call *drop-out batching*. There, packets are flagged and remain in the batches ensuring optimal performance for the fast path. At the same time, flagged packets are inserted into a separate queue enabling efficient batch processing for the slow path.

5.3.2 Flexible Architecture

The two primary goals of MoonRoute are its high flexibility and easy extensibility. MoonRoute does not use DPDK directly but relies on libmoon, presented in Section 3.2, inheriting its programming language and compiler—Lua and LuaJIT. Modules provide the typical functionality of the router. These modules represent a specific step in the packet processing task, such as the routing table lookup, and take an array of packet buffers and a vector of flags as input. An optional parameter for these modules is the queue to a slow path component, where packet buffers can be inserted by the module if necessary. Such a module returns the array of packet buffers and an output vector of flags, signaling the packets to be processed in the next step.

The supported languages to write modules are C/C++ and Lua. The usage of the Lua language is intended for the initial implementation of quick, low-effort prototypes. It is possible to switch the prototype implementation for high-performance C/C++ libraries at a later point in time or reuse existing libraries conveniently through LuaJIT's integrated FFI. The included reference router contains a Lua-written main loop, which connects all modules of the router. Adding Lua code extensions to this main loop is simple. Most of the performance-critical functionality is handled by C libraries included in DPDK. The libraries are wrapped by a lightweight Lua wrapper calling the C library via the FFI. MoonRoute's default router only supports IPv4 in its current version.

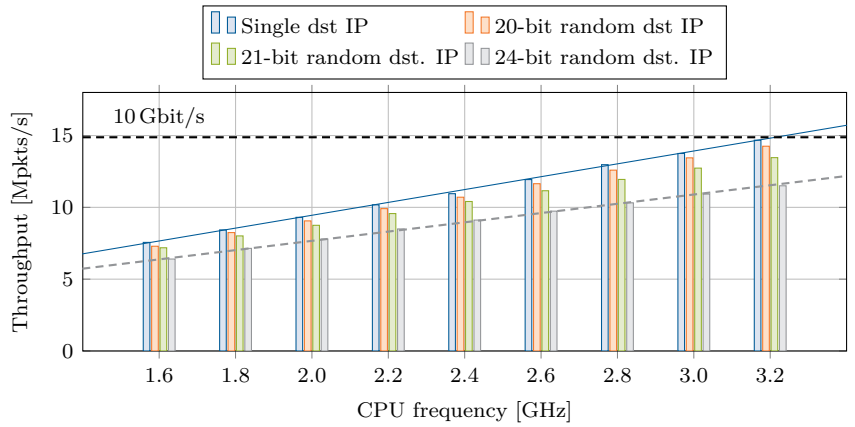
5.3.3 Evaluation and Modeling

Upcoming many-core CPU architectures make parallelization and scalability the critical aspects of high-performance designs. The following measurements demonstrate how the default router implementation of MoonRoute scales with CPU clock and cores.

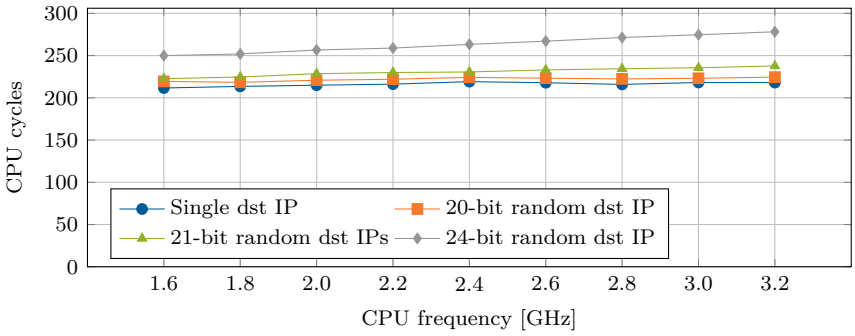
Scaling with CPU frequency

The measurements in Sections 5.1 and 5.2 focus on the underlying frameworks that are used as a part of packet processing applications. This section investigates an implementation of a software router resembling a real-world application—the MoonRoute framework. For the following evaluation, we use MoonRoute's default router implementation. The test servers are equipped with an Intel E3-1230 CPU with a base frequency of 3.2 GHz. The CPU has core-exclusive L1 and L2 caches with 32 kB and 256 kB and a shared LLC of 8 MB. Further, the server features an Intel X540-T2 NIC offering a bandwidth of 2×10 Gbit/s.

Figure 5.7 shows the scaling of MoonRoute's default router with the CPU frequency under different test scenarios. The test scenarios were chosen to test the router with a different number of routing table entries. The routing table utilized by MoonRoute implements the DIR-24-8 algorithm described by Gupta et al. [82]. This algorithm performs the longest prefix match



(a) Throughput for different CPU core frequencies (cf. Emmerich et al. [81])



(b) Per-packet costs for different CPU core frequencies

Figure 5.7: Scaling of MoonRoute with CPU frequency

as a two-step process. IP addresses are matched with a single lookup if their subnet has a prefix length of /24 or shorter; otherwise, two lookups are required. Our measurements focus on measuring the performance of the single-step lookup, so the control flow of the algorithm is not changed due to longer prefixes during the measurement. The data structure used for the first lookup in DIR-24-8 is an array with 2^{24} 16-byte entries, resulting in a size of 32 MB.

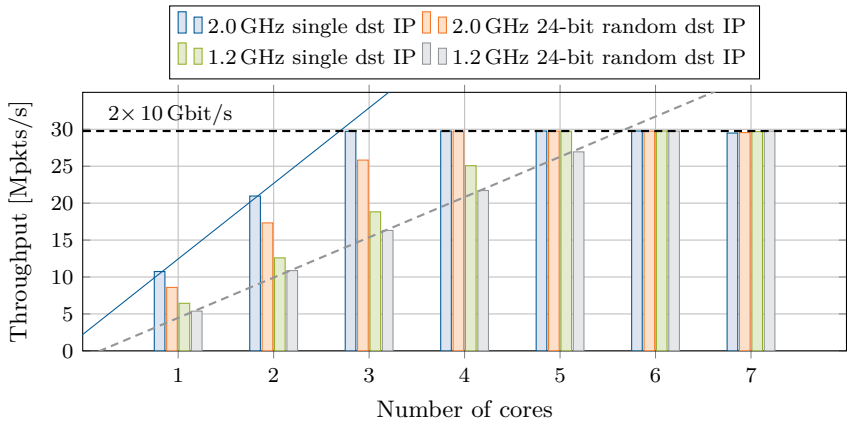
Figure 5.7a demonstrates that performance is not bandwidth-bound, but CPU-bound for all investigated scenarios. The per-packet costs for each scenario are displayed in Figure 5.7b. These costs can be classified into different components, as previously defined in Equation 5.4. With throughput being CPU-bound c_{busy} is 0. We already measured the constant costs of 100 cycles on DPDK for c_{IO} in Section 5.1.5. The remaining costs are part of c_{task}. A router causes different kinds of CPU load for c_{task}: processing $c_{task-calc}$ and lookup $c_{task-access}$. The component $c_{task-calc}$ sums up all tasks the router performs, which are not lookup-related, e.g., checking packet headers or decrementing header fields. Performing the longest prefix match itself is contained in $c_{task-access}$. This leads to Equation 5.8 for the routing costs per packet:

$$c_{routing}^{packet} = c_{IO} + c_{task-calc} + c_{task-access} \tag{5.8}$$

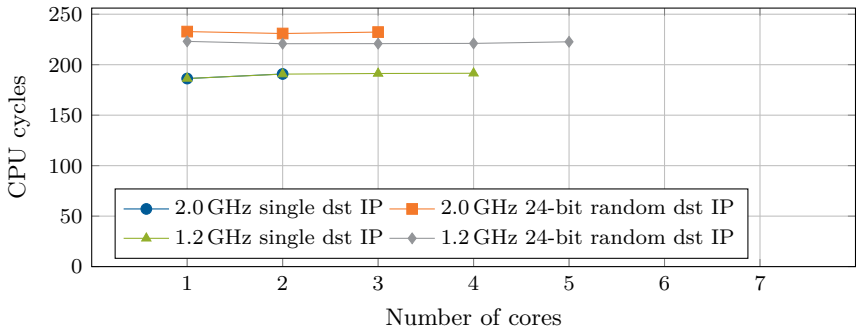
The $c_{task-calc}$ component is constant for all packets, as the processing costs for each packet are the same. The $c_{task-access}$ component, however, depends on the data access costs. Figure 5.7b demonstrates different scenarios. The first scenario uses test traffic with only a single destination IP address. Therefore, the same address is looked up in the routing table on each access. This is the optimal scenario from a caching perspective; the same 16-byte entry is accessed repeatedly. The accessed date is therefore put into the fastest cache available, minimizing access costs. The next scenario uses an IP destination address where the 20 most significant bits are randomized. This means that 2^{20} different entries are looked up repeatedly over the measurement period in the routing table. Every entry contains 2 B of information. During the measurement, the CPU tries to cache the accessed entries, which results in a total amount of 2 MB being cached. This amount of data uses the caches up to the LLC. The next scenario increases the number of randomized, most significant bits in the destination address to 21. This doubles the amount of used data to 4 MB, putting even more stress on the LLC. The final, most demanding scenario increases the amount of used data to 32 MB. This exceeds the limit of the LLC, leading to matches requiring access to RAM.

Figure 5.7b shows roughly constant per-packet costs across the investigated CPU frequencies for up to 21 bit. However, in the case of the 24 bit, per-packet costs rise by approximately 40 cycles. This is a consequence of the two different components $c_{task-calc}$ and $c_{task-access}$. If the CPU frequency is doubled from 1.6 GHz to 3.2 GHz, the scenario with the single address also doubles in throughput. This means that per-packet costs, which are dominated by $c_{task-calc}$, remain constant. If the CPU frequency is doubled from 1.6 GHz to 3.2 GHz, the scenario with the 24-bit randomized addresses does not double in throughput. The per-packet costs increase, which have a larger amount of $c_{task-access}$. The reason for this sub-linear behavior is that increasing clock frequencies of the CPU do not accelerate the time needed for accessing data, leading to the observed behavior.

Knowing the per-packet costs and the available number of CPU cycles, the throughput performance of MoonRoute can be predicted. For small routing tables, where the influence of routing table access costs or $c_{task-access}$ are low, constant per-packet costs lead to a good approximation of the throughput. Between the least demanding (1 address) and the most demanding scenario (2^{22}), there is a 5 % difference in per-packet costs. However, in the case of large routing tables requiring RAM accesses, an approximation becomes more difficult. We cannot give a precise prediction without knowing the access pattern created by the received traffic. If the test traffic is known, measurements can help to determine the routing table access costs. With these access costs, an appropriate prediction of per-packet costs and subsequently throughput performance becomes possible.



(a) Throughput for different number of CPU cores (cf. Gallenmüller et al. [80])



(b) Per-packet costs for different CPU core frequencies

Figure 5.8: Scaling of MoonRoute with the number of CPU cores

Scaling across CPU cores

Figure 5.8 shows the scaling with the number of CPU cores. In this measurement, we switched from a 4-core to an 8-core CPU to test scalability across a higher number of cores. We use an Intel 2.0 GHz Xeon E5-2640 v2 CPU, with the same L1 and L2 cache sizes but a 20 MB LLC. In addition, we use bidirectional traffic to increase the non-bandwidth-bound range of the DuT.

Figure 5.8a shows a clear linear trend for scaling across multiple cores. Figure 5.8b visualizes the per-packet costs of this measurement. CPU cycles spent on busy polling may distort the per-packet costs. Therefore, we calculate the per-packet costs where no busy polling happens ($c_{busy} = 0$), i.e., where the router is fully loaded. Per-packet costs demonstrate that the scaling is perfect when considering a scenario with only a single destination IP address. The per-packet costs are constant at 190 cycles, independent of the core clock rate or the number of cores. In the scenario where the 24 most significant bits are randomized, there is a difference between the different clock rates, with the higher clock rate having higher packet costs. At 1.2 GHz, we measure per-packet costs of 220 cycles and 230 cycles at 2.0 GHz. This difference can be attributed to the data access costs ($c_{task-access}$), which increase as cache and RAM accesses are not scaled with the CPU core frequency.

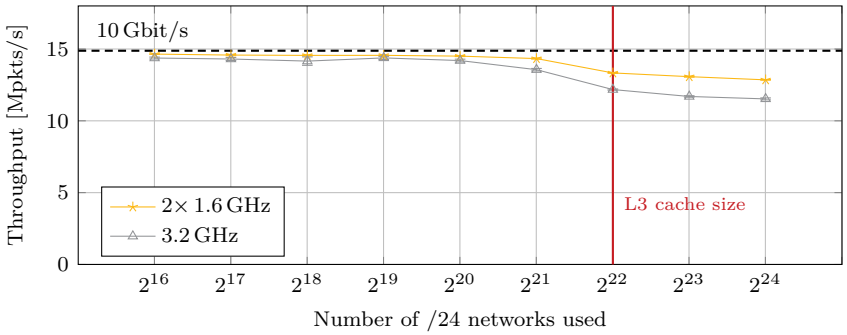


Figure 5.9: Comparison of LPM execution on one and two CPU cores (cf. Emmerich et al. [81])

Figure 5.8b does not show rising per-core costs as the 24-bit plot shown in Figure 5.7b. The measurements use CPUs that offer different LLC sizes (8 MB vs. 20 MB). The larger LLC hides the high costs for accessing RAM.

Our measurements show that scaling across CPU cores works well for MoonRoute, offering almost perfect linear scaling with up to 6 CPU cores. Without cache effects, the throughput can be predicted by using the available CPU cycles on several cores and by measuring the components of the per-packet costs c_{IO} and $c_{task-calc}$. Cache effects, reflected by $c_{task-access}$, increase costs and are not easily predictable without measuring a specific scenario. However, measurements have shown that a large LLC can help to lower the impact of RAM access on throughput performance.

Frequency vs. Per-Core Scaling

The measurement depicted in Figure 5.9 uses the Intel E3-1230. There, the two scaling methods, CPU cores and frequency, are compared directly. The first scenario uses two cores throttled to a frequency of 1.6 GHz; the second scenario uses a single core with a clock speed of 3.2 GHz. Although the same number of CPU cycles is available in both setups, the two-core scenario outperforms the single-core scenario for every investigated routing table size. The relative outperformance for the two-core scenario starts at a value of 2 % and increases to 12 % after the routing table size surpasses the capacity of the LLC.

The difference between both setups is the availability of lower-level caches. Both cores in the two-core setup have their exclusive L1 and L2 caches. With RSS in place, the addresses are partitioned between the two cores; therefore, the chance for a cache hit increases. Both effects, the larger amount of lower-level cache and the more efficient cache usage, lead to the outperformance of the two-core scenario. Using the notation of the model, $c_{task-access}$ is lower for the two-core scenario.

Performance prediction is again highly dependent on the access scenario. However, routing performance profits from the availability of fast caches, therefore a configuration using more cores and more importantly, a larger amount of fast caches can have a positive performance impact.

Router	Throughput	
	[Mpkts/s]	[%]
MoonRoute	14.6	100
FastClick (DPDK 2.2)	10.4	72
Click (DPDK 2.2)	4.3	29
Linux 3.7	1.5	10

Table 5.1: Single-core router performance (cf. Gallenmüller et al. [80])

Comparison to Other Software Routers

Table 5.1 demonstrates the single-core performance of MoonRoute compared with other software routers, such as the Linux Router, the modular software router Click and FastClick, both using DPDK as backend. FastClick [59] extends Click with performance-enhancing techniques, e.g., batch processing. MoonRoute can almost saturate a 10 Gbit/s link utilizing a single core, improving performance between 30 and 90 % compared with its contestants. These tests demonstrate the optimal throughput for the respective software routers, each containing only a single routing table entry.

5.3.4 Conclusion

Modularity and high performance—often considered conflicting goals for optimization—are achieved by MoonRoute's careful design choices: the two-path design separating high from low priority tasks, improved performance with batching techniques, and multi-thread optimized data structures. The just-in-time compilation for Lua allows leveraging the flexibility of a scripting language without sacrificing performance and employing a convenient FFI to allow easy code reuse. The performance evaluation of our reference router offers insights into its scalability. Routing costs can be divided into two classes, the data access costs and processing costs. We measured that throughput performance scales perfectly linear with the CPU frequency and across several CPU cores when considering only the processing costs. This allows a reliable prediction of throughput performance. However, data access cannot be scaled as easily, which may lead to sub-linear scalability. The impact of the data access costs on throughput performance depends on the specific usage scenario. Measurements of this usage scenario can help to determine access costs reliably, thus allowing a throughput prediction. Scaling across CPUs can additionally increase the amount of cache memory, which helps to minimize data access costs.

5.4 Ultra-Reliable Low-Latency Communication

Section 5.4 is joint work between Sebastian Gallenmüller, Johannes Naab, Iris Adam, and Georg Carle [4].

The flexibility and adaptability of 5G are considered its main features, enabling the creation of dedicated wireless networks, customized for specific applications with a certain level of QoS. The International Telecommunication Union (ITU) identifies three distinct services for 5G networks [83]: *enhanced mobile broadband* (eMMB), a service comparable to LTE networks optimized for high throughput; the *massive machine type communication* (mMTC), a service

designed for spanning large IoT networks optimized for a large number of devices with low power consumption; and the *ultra-reliable low-latency communication* (URLLC), a service for safety-critical applications requiring high reliability and low latency. These different services can be realized by *slicing* the network into distinct, independent logical networks, which can be offered as a service adhering to customer-specific SLAs, called Network Slice-as-a-Service. A cost-efficient way to realize network slices is the shared use of network resources among customers, e.g., virtualization techniques used on off-the-shelf servers. This makes virtualization and its implications on performance one of the crucial techniques used for 5G. Virtualization is the natural enemy of predictability and low latency [84], posing a significant obstacle when realizing URLLC. In this thesis, we investigate if and how the seemingly contradictory optimization goals, virtualization and resource sharing on the one side and low latency and high predictability on the other side, can go together. The goals of our investigation are threefold:

1. creating a low-latency packet processing architecture for security functions with minimal packet loss,
2. conducting extensive measurements applying hardware-supported timestamping to precisely determine worst-case latencies, and
3. introducing a model to predict the capacity of our low-latency system for overload prevention.

Our proposed system architecture relies on well-known applications and libraries, such as Linux, DPDK, and Snort. Besides the specific measurements for the Snort IPS, we investigate the performance of the underlying OS and libraries in use, namely Linux and DPDK, which emphasizes that our results are not limited to Snort but are highly relevant to other low-latency packet processing applications.

The remainder of our investigation is structured as follows: Section 5.4.1 demonstrates the need for a new system design of security functions. Background and related work are presented in Section 5.4.2. In Section 5.4.3, we describe our novel system architecture that is evaluated in Section 5.4.3. In Section 5.4.4, we present our model for overload prediction. Considerations about the limitations and the reproducibility of our system architecture are given in Sections 5.4.5 and 5.4.6. Finally, Section 5.4.7 concludes our analysis by summarizing the most relevant findings and proposes enhancements for future work.

5.4.1 Motivation

The ITU [83] defines the requirements for the URLLC service as follows: the one-way delay of 5G radio access networks (RAN) from source to destination must not exceed 1 ms and the delivery success rate must be above 99.999 %. We demonstrate how security functions facing input/output events, interrupts, and CPU frequency changes behave concerning the challenging URLLC requirements.

The following example uses Snort as an inline intrusion prevention system, i.e., every packet has to pass through Snort, which subsequently influences the delay of every packet. For this example, all filtering rules are removed, turning Snort into a simple forwarder that is not influenced by

	n-*th* percentiles				
	50	99	99.9	99.99	99.999
Snort-fwd	69 µs	88 µs	107 µs	1.7 ms	2.5 ms

Table 5.2: Latencies of a Snort forwarder (cf. Gallenmüller et al. [4])

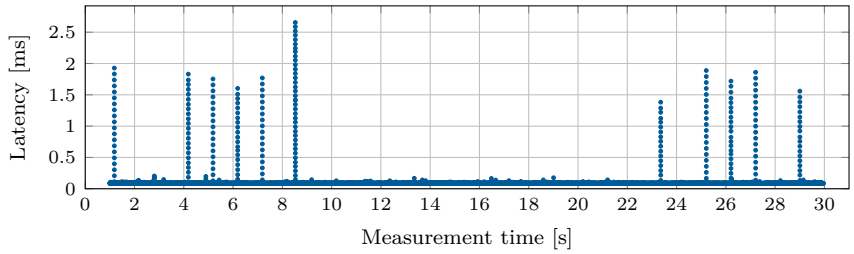


Figure 5.10: Snort forwarder worst-case latencies (cf. Gallenmüller et al. [4])

any rule processing. Therefore, the observed behavior represents a best-case scenario providing a lower latency bound for Snort IPS execution. Snort runs in a Virtual Machine (VM), providing a realistic multi-tenant setup for 5G networks. The packet rate is set to 10 kpkts/s, a moderate system load without any packet drops. The measurement runs 30 s. As we are not interested in the latency spikes caused by the application start-up, we exclude the first second of measurements from Figure 5.10 and Table 5.2.

Table 5.2 shows the percentiles of latency we measured. Up to the 99.9th percentile, the observed latency budget is low enough to allow additional packet processing tasks while still meeting URLLC requirements. For higher percentiles, the basic forwarder already exceeds the latency budget. Figure 5.10 shows a scatter plot displaying the 5000 worst-case latencies measured over 30 s. We see that latencies exceeding the 1 ms latency budget are not only occurring at the beginning of measurements due to cache warm-up or other ramp-up effects, but in an irregular and unpredictable pattern throughout the entire measurement. The latency spike pattern did not change over time. Therefore, we consider this being the steady-state behavior of our investigated application.

Thus, different system designs for security functions are needed to meet the strict requirements of URLLC services. In the following, we demonstrate techniques and frameworks, creating a low-latency software stack to show that meeting URLLC requirements is possible while using the same hardware as for this motivating example.

5.4.2 Background and Related Work

After introducing the challenges of 5G, this section focuses on techniques impacting the delay and jitter caused by software packet processing systems.

URLLC in Industry 4.0: 5G has a strong focus on critical infrastructures or industrial applications, for instance, industrial control applications of cyber-physical systems. These applications introduce new requirements for cellular networks, e.g., a high level of availability and end-to-end

realtime support [85]. According to Yoshizawa et al. [86] 5G, and URLLC especially, must be developed and designed with security in mind, protecting the user equipment and the network infrastructure from potential attacks. Our target is the definition of an architecture and mechanisms for security (monitoring) functions to guarantee the QoS during design, deployment, and modification for URLLC use cases.

Polling vs. interrupts: One possible cause for OS interrupts is the occurrence of IO events, e.g., arriving packets, to be handled by the OS immediately. Interrupt handling causes short-time disruptions for currently running processes. The ixgbe network driver and Linux employ moderation techniques to minimize the number of interrupts and the influence on processing latency [87]. Both techniques were introduced as a compromise between throughput and latency optimization. For our low-latency design goal, neither technique is optimal, as the interrupts—although reduced in numbers—cause irregular variations in the processing delay, which should be avoided. DPDK [47], a framework optimized for high-performance packet processing, prevents triggering interrupts for network IO entirely. It ships with its own userspace driver, which avoids interrupts but polls packets actively instead. Avoiding interrupts leads to execution times with only little variation also due to DPDK's preallocation of memory and a lack of costly context switches between userspace and kernelspace. However, polling requires the CPU to wake up regularly, increasing energy consumption. The Linux kernel's Express Data Path (XDP) offers similar capabilities to DPDK [88]. We focus on DPDK as a DPDK-enabled IPS is already available [89].

CPU features: Numerous guides list CPU and OS features, leading to unpredictable behavior for application performance on which the following recommendations are based on [90]–[92]. HyperThreading (HT) or simultaneous multithreading (SMT) is a feature of modern CPUs that allows addressing physical cores (p-cores) as multiple virtual cores (v-cores). Each p-core has its own physically separate functional units (FU) to execute processes. However, multiple v-cores are hosted on a p-core, sharing FUs between them. Zhang et al. [93] demonstrate that sharing FUs between v-cores can impact application performance when executing processes on v-cores instead of the physically separate p-cores. Modern CPUs can be switched into different sleep states, which lower CPU clock frequency and power consumption. Switching the CPU from an energy-saving state to an operational state leads to wake-up latencies. Schöne et al. [94] measured wake-up latencies between 1 and 40 μs for Intel CPUs depending on the state transition and the processor architecture.

Despite having physically separate FUs, p-cores share a common LLC. Therefore, processes running on separate p-cores can still impact each other competing on the LLC. Herdrich et al. [95] observed a performance penalty of 64 % for a virtualized, DPDK-accelerated application when running in parallel with an application utilizing LLC heavily. The uncontended application performance can be restored for the DPDK application by dividing the LLC statically between CPU cores utilizing the cache allocation technology (CAT) [95] of modern Intel CPUs.

OS features: Beside interrupts caused by IO events, an OS uses interrupts for typical tasks, such as scheduling or timers. Patches for the Linux kernel [96] were introduced to create a more

predictably behaving kernel, e.g., by reducing the interrupt processing time. Major distributions, such as Debian, provide this, so-called PREEMPT_RT kernel, as part of their package repository. Besides, the Linux kernel offers several command-line arguments influencing latency behavior. Cores can be excluded from the regular Linux scheduler via `isolcpu`. Isolated CPU cores should be set to `rcu_nocb`, lowering the number of interrupts for the specified cores.

Low-latency VM IO: Transferring packets into/out of a VM leads to significant performance penalties compared with bare-metal systems. In previous work [84], we compared packet forwarding in bare-metal and VM scenarios, demonstrating that VMs can introduce high tail latencies of 350 μs and more. We demonstrated that DPDK can help improve forwarding latencies but must be used on the host system and the VM. Furthermore, modern NICs supporting single root IO virtualization (SR-IOV) can be split into several independent virtual functions, which can be used as independent NICs and can be bound to VMs exclusively. In this case, virtual switching is done on the NIC itself, minimizing the software stack involved in packet processing. In an investigation by Lettieri et al. [97], SR-IOV, among other techniques for high-speed VM-based NFs, is one of the fastest techniques with the lowest CPU utilization. Therefore, the latency performance of SR-IOV is superior to software switches, e.g., Xu and Davda [98] measured an almost 10-fold increase of worst-case latencies for a software switch. Xiang et al. [99] create and evaluate an architecture for low-latency NFs. Their architecture provides sub-millisecond latencies, but they do not investigate the worst-case behavior. Zilberman et al. [100] give an in-depth latency analysis of various applications and switching devices. They stress the need for tail-latency analysis to analyze application performance comprehensively.

The topic of VM-based NFs has been extensively researched in literature [97]–[99]. However, given our motivating example in Section 5.4.1 and the importance of the URLLC service, we argue, similar to Zilberman et al. [100], that the crucial worst-case behavior needs close attention. Hence, we aim to create the lowest latency system achievable, utilizing available applications on off-the-shelf hardware.

There are also embedded systems such as jailhouse [101] or PikeOS [102] being able to partition the available hardware providing realtime guarantees for user processes or VMs. However, they are either not compatible with standard Linux interfaces such as libvirt or replace the host OS entirely. Therefore, the tool support for these specialized hypervisors is worse than the more widespread solutions such as Xen or Kernel Virtual Machine (KVM) utilizing the libvirt software stack. Thus, we do not consider these specialized solutions for this work but rely on well-established software tools and hardware.

5.4.3 System Architecture

We investigate the performance of basic components such as RT Linux, KVM, DPDK, SR-IOV, and CAT. We demonstrate the system architecture of a single server that uses these basic components to run low-latency applications. Our results are also relevant to large-scale cloud deployments comprised of the same components, such as OpenStack.

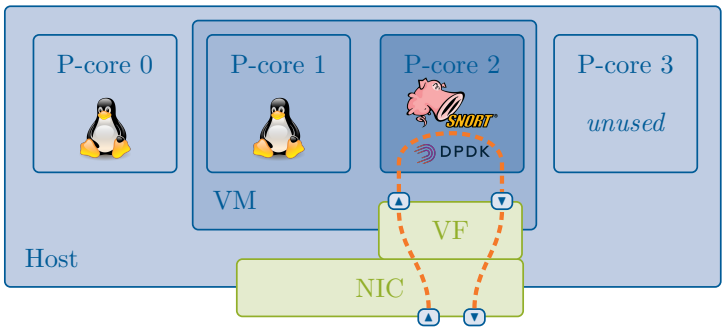


Figure 5.11: System architecture overview (cf. Gallenmüller et al. [4])

Figure 5.11 shows the structure of our low-latency VM running on a CPU with four physical cores (p-cores), following the optimizations presented in Section 5.4.2: disabling SMT to avoid any influence of v-cores, using a PREEMPT_RT kernel on the host and the VM to minimize interrupt latencies for the virtualized packet processing application, and utilizing core isolation to dedicate cores to specific processes to minimize the QoS impact between cores and applications running on them. The OS of the *host* is restricted to p-core 0, isolating p-cores 1 and 2 for exclusive VM usage. On the *VM* itself, the OS runs on p-core 1 exclusively, isolating p-core 2. P-core 2, isolated from host and VM, runs DPDK and Snort. The core isolation feature complements DPDK's design philosophy of statically pinning packet processing tasks to cores. Utilizing SR-IOV, the NIC is split into virtual functions (VF). One VF is passed through to the VM attached to p-core 2. The critical network path and its associated CPU resources are isolated from OS tasks providing a stable service for latency-critical processes. We disable the energy-saving states in the BIOS or set them to the most reactive state to avoid any delays caused when waking up the CPU. We use Intel CAT to statically assign a certain amount of the LLC to p-core 2. USB legacy emulation and system management interrupts should be disabled in BIOS if possible.

Evaluation

The following measurement series characterizes the latency behavior of the proposed architecture.

Setup

Figure 5.12 shows the setup used for testing based on three machines. The DuT runs Snort, forwarding traffic between its physical interfaces, the other two machines run the packet generator MoonGen [16]. The load generator (LoadGen) acts as traffic source/sink generating/receiving the test traffic, the third machine (timestamper) monitors the entire traffic received/sent by the DuT. The timestamper monitors the traffic via passive optical Terminal Access Points (TAPs), timestamping every packet in hardware with a 12.5 ns resolution [19]. Being passive, the optical TAPs do not introduce variation to the timestamping process. Timestamping every single packet only works for receiving ports. Therefore, we timestamp on a separate host instead of the LoadGen itself.

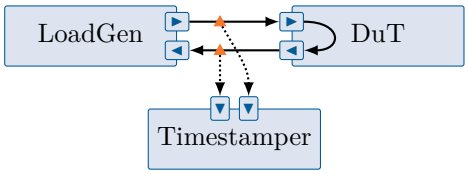


Figure 5.12: Setup with Snort as a DuT, MoonGen as a LoadGen, and a Timestamper (cf. Gallenmüller et al. [4])

The three servers are equipped with Supermicro mainboards (X10SDV-TP8F) featuring Intel Xeon D-1518 CPUs (4 cores, 2.2 GHz) and an onboard Intel X552 NIC (dual-port SFP+, 10G Ethernet). On the DuT, we use Debian Buster (kernel v4.19) as OS, KVM as hypervisor, and the current beta of Snort (v3.0.0) [103] together with a DPDK-enabled data acquisition plugin (daq, v2.2.2) [89]. Section 5.4.6 lists the repositories, commit ids of the investigated applications, configuration data, and used measurement tools. The VM configuration is shown in Figure 5.11. Via Intel CAT [95], we pin 4 MB of the LLC to the core running the packet processing application; the remaining 2 MB are shared among the other cores.

We opt for UDP-only test traffic to prevent TCP congestion control from impacting the measured latency. The UDP destination port is set to 53 to trigger Snort's rules for DNS processing. The payload of our generated traffic does not contain DNS information but a counter to efficiently track packet loss and forwarding latency. We use CBR traffic for testing and dedicate Section 5.4.3 to measure the impact of bursty traffic.

Measurements

The following measurements investigate the performance of our proposed architecture regarding the URLLC requirements. Therefore, we aim for a packet delivery rate above 99.999 % and a latency below 1 ms. We do not replicate the entire end-to-end communication path of 5G, but only a security function located in the 5G backend network; therefore, we aim for a lower latency goal. Faced with a similar problem, Xiang et al. [99] calculated a latency goal of 350 μs for their NFC. In this thesis, we apply the same latency goal to our measurements, quantifying the performance of Snort. We try to isolate the influence of the IO framework (DPDK), Snort overhead, and rule processing through separate measurements. Therefore, we test three related packet forwarding applications:

1. *DPDK-l2fwd*, being the most simple forwarder in our comparison, representing the minimum latency of IO without any processing happening;
2. *Snort-fwd*, forwarding packets with Snort on top of DPDK, which quantifies the overhead caused by Snort without any traffic filtering happening; and
3. *Snort-filter*, applying the Snort 3 community ruleset [104] to the forwarded traffic. The filter scenario does not drop any packet because we are only interested in the overhead caused by rule application.

We measure between 10 and 120 kpkts/s incremented in 10 kpkts/s steps. Due to space limitations, we only show three selected rates for every scenario in Table 5.3. For each scenario, we

	Mode	Rate [kpkts/s]	Loss [%]	n-*th* percentiles 50 [μs]	99 [μs]	99.9 [μs]	99.99 [μs]	99.999 [μs]	Max. [μs]
DPDK-l2fwd	HW	10	-	3.1	3.4	7.7	12.2	13.4	13.6
	HW	60	-	3.1	3.3	8.3	13.5	14.4	16.0
	HW	120	-	3.1	3.3	8.1	13.2	14.3	14.6
	VM	10	-	3.3	4.0	14.7	17.6	19.0	19.2
	VM	60	-	3.3	4.0	15.4	18.9	19.9	21.5
	VM	120	-	3.3	3.9	16.6	20.2	21.3	22.9
Snort-fwd	HW	10	-	14.5	24.7	29.7	32.4	33.1	33.1
	HW	80	0.1	14.4	29.9	43.7	46.2	47.7	50.6
	HW	90	3.3	30 609.5	30 834.8	30 882.7	30 915.3	30 936.1	30 959.1
	VM	10	-	15.9	37.6	58.6	66.8	68.0	68.6
	VM	80	0.1	18.8	73.9	98.8	115.6	117.7	121.9
	VM	90	7.1	2469.6	2657.9	2679.8	2692.2	2700.6	2708.3
Snort-filter	HW	10	-	17.4	28.2	33.1	35.8	36.4	36.6
	HW	60	0.0	17.1	29.0	34.1	36.1	50.4	51.5
	HW	70	0.0	79.0	24 897.2	27 521.2	27 847.0	27 947.1	27 992.9
	VM	10	-	18.4	40.9	63.1	71.8	73.4	73.7
	VM	60	0.1	17.5	62.2	92.9	101.1	114.7	115.7
	VM	70	3.0	3036.9	3270.2	3294.4	3313.1	3326.8	3342.5

Table 5.3: Latencies of different software systems (cf. Gallenmüller et al. [4])

list the minimal rate of 10 kpkts/s, the last rate before overloading the DuT, and the first rate when the DuT has been overloaded. The actual packet rates depend on the individual scenario. Being able to process millions of packets per second without overloading, we could not overload the DPDK forwarder within the selected packet rates [38]. Therefore, we present 10, 60, and 120 kpkts/s representing low, medium, and maximum load in this case.

Hardware

Initially, we test the forwarding applications in a non-virtualized setup to measure the performance baseline (cf. Table 5.3, mode: HW).

DPDK-l2fwd: We measure the behavior of the DPDK forwarder for packet rates of 10, 60, and 120 kpkts/s. The median forwarding latency is 3.1 μs and increases slightly to a maximum of 3.4 μs for the 99th percentile indicating a stable latency behavior. Only the rare tail latencies, i.e., above the 99.9th percentile, increase to a maximum value of 16.0 μs. The overall latency values do not differ significantly between measurements. We did not observe any packet loss for the three tested rates.

Snort-fwd: Running Snort on top of DPDK increases latency significantly. The median for rates of 10 and 80 kpkts/s is almost the same with 14.5 and 14.4 μs, respectively. This new median is almost as high as the worst-case latency for the DPDK forwarder. Tail latencies increase further and seem to depend on the packet rate, i.e., tail latencies increase for higher packet rates. At a rate of 80 kpkts/s, packet drops can occur. A closer analysis shows that a consecutive sequence of packets is lost only at the beginning of the measurement, despite the previous warm-up

phase. As packet loss does not occur later, we do not consider this configuration as an overload scenario. We consider the rate of 90 kpkts/s as an overload scenario, which is characterized by the noticeable packet loss (3.3 %) and the over thousandfold latency increase compared with the median latency of the previous measurements. The latency increase in the overloaded scenario results from packets not being processed fast enough, leading to buffers filling up. Therefore, the worst-case latency remains at this high level for all observed percentiles.

Snort-filter: For this measurement, the Snort forwarder applies the community ruleset. Rule application introduces additional costs resulting in a latency offset of roughly 3 μs compared with the previous measurement at 10 and 60 kpkts/s. Only the worst-case latencies differ noticeably for the latter. The overload scenario already occurs at a lower rate of 70 kpkts/s due to the higher processing complexity indicated by the high tail latencies. Loss rates and median would still be tolerable. However, the tail latencies show an increase by a factor of over 1000 compared with the median. When comparing the load scenarios before overloading, Snort filter processes packets with lower latency than its respective counterpart for the Snort forwarder. We attribute this to the relative load, which is higher for the Snort-fwd, i.e., it is more overloaded at 90 kpkts/s than the Snort-filter at a rate of 70 kpkts/s.

Virtualization

Processing packets in virtualized environments can have a significant impact on latency. To measure the impact, we repeat the previous measurements in a virtualized environment (cf. Table 5.3 (Mode: VM), Figure 5.11).

DPDK-l2fwd: In the virtualized environment, latency increases compared with the non-vitualized measurements. The median latency increases by 6 %, but the tail latencies can increase by almost 60 %. Table 5.3 shows that, up to the 99th percentile, the packet rate has little influence on latency. For higher percentiles, a trend towards higher latencies seems to manifest.

Snort-fwd: Compared to its hardware counterpart, latency increases by 30 % for the median and up to more than 100 % for the tail latencies. We observe the same initial packet loss in the non-overloaded case. Packet loss in the overloaded case is higher, as packet processing is more expensive in the VM for the same packet rate. In the overloaded case, latencies are over ten times lower than the hardware measurements, still violating the 1 ms goal. Measurements show that enabling SR-IOV leads to the decrease for worst-case latencies due to smaller buffers, an observation confirmed by Bauer et al. [105].

Snort-filter: Comparing Snort-filter in virtualized and non-virtualized environments shows that the median latency increase is below 1 μs. The values for the tail latencies increase by a factor of two or more for the virtualized environment.

Looking at Table 5.3, we can conclude that URLLC-compliant latency is only violated if the DuT is overloaded. Overload latencies rise by a factor of 1000 for the HW scenario and by a factor of 100 for the VM scenario. Without overloading the system, the latencies are below URLLC requirements, even for the most challenging scenario. When considering the worst-case

scenario, Snort-filter (HW), we measure a latency of 50.4 μs at the URLLC-required 99.999th percentile. The overall observed worst-case latency for the VM scenario for the 99.999th percentile is 117.7 μs. Despite the latency difference of over 100 % between HW and VM, both worst-case scenarios—HW and VM—do not violate our latency goal of 350 μs. In fact, the remaining latency budget allows for even more complex packet processing tasks.

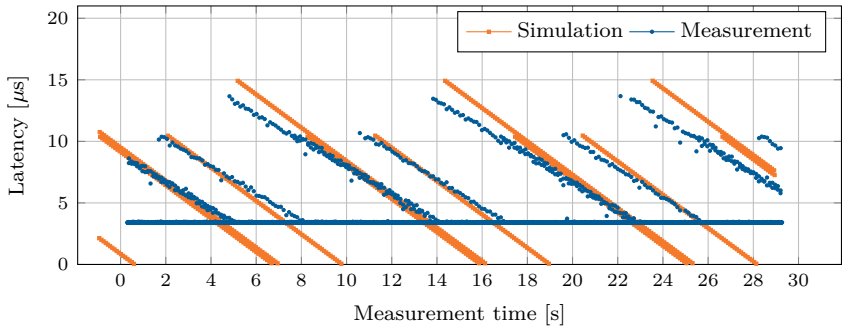
Tail latencies

We have previously shown that the measured tail latencies in the non-overloaded scenario do not impair URLLC latency goals. In this section, we want to investigate the effects causing the tail latencies to exclude potentially harmful consequences such as latency spikes or even short-term overload. Increased tail latencies are already present in the DPDK-l2fwd scenario, indicating that their causes are already part of the basic packet processing steps. We investigate the differences between bare-metal deployment and virtual environment.

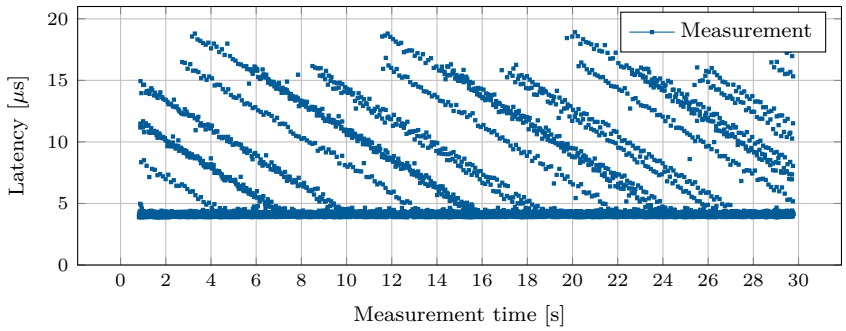
HW: We analyze the tail latencies in the non-overloaded scenarios. Figure 5.13a shows a scatter plot of the 5000 highest latency events measured over 30 s. The figure shows a horizontal line at approximately 3.4 μs, the area where the majority of latency events happen, which matches the 99th percentile given in Table 5.3, Line 2. Above this horizontal line, a regular linear pattern over the 30-second measurement period is visible. We assume that the latency events above the horizontal line of 3.4 μs are a result of packets being delayed due to interrupt processing in the OS. To investigate our assumption, we record the interrupt counters (`/proc/interrupts`) of the OS during the measurement.

We identified the pattern above the horizontal line as an interplay of two clocked processes—OS interrupt generation on the DuT and generated CBR traffic pattern on the LoadGen. The observed pattern is created by an effect known as aliasing. Here, we use the generated traffic as a sampling process, trying to detect interrupts. As the interrupts are too short ($\leq 13.6\,\mu$s) to be correctly detected at the generated traffic rate of 10 kpkts/s (100 μs inter-packet gap), we undersample leading to the observed pattern. The OS interrupt counters (`/proc/interrupts`) revealed local timer interrupts (loc) and IRQ work interrupts (iwi) to be the only interrupts triggered on the packet processing core of the DuT during operation. We measured, using the TSC of our CPU, constant execution times of 8.2 μs for the iwi and 5.5 μs for the loc. The two different execution times are visible in Figure 5.13a as longer and shorter lines. Their maximum values of 10.9 and 13.6 μs differ because additional tasks such as packet IO and context switches are included. Packets are generated at a rate of 10 kHz, and we measure interrupts being generated at a rate of 250 Hz. Locs and iwis happen in a regular pattern; an iwi is triggered after every second loc.

To verify whether the interrupts cause the observed pattern, we create a script simulating the described process using the measured frequencies and processing times. Figure 5.13a shows similar patterns for the simulation confirming our assumptions. Measurement and simulation are highly sensitive to the maximum measured values, the traffic rates, and the interrupt rate. Even minor parameter changes, e.g., restarting the load generator, can lead to changes in the generated traffic rate and, therefore, lead to different patterns. The same happens if the traffic



(a) HW (cf. Table 5.3, Line 2)



(b) VM (cf. Table 5.3, Line 5)

Figure 5.13: 5000 worst-case latency events measured for DPDK-l2fwd at 10 kpkts/s (cf. Gallenmüller et al. [4])

rate is increased or lowered. This sensitivity means that repeating the same measurements may lead to patterns with different shapes and orientations. However, a regular pattern can be observed as long as the interrupt process is undersampled.

VM: Figure 5.13b shows the 5000 highest latency events measured for the DPDK-l2fwd (VM) scenario. The entire graph is shifted, the horizontal line is shifted to approximately 4.4 µs, the long interrupt latency is approximately 19 µs, the shorter approximately 16 µs, indicating the higher overhead when running in a VM. The number of events above the horizontal line roughly doubled. This increase can be explained by the fact that now two OS (VM host and VM) trigger interrupts. We observed the same interrupts for the VM host as in our HW measurement. For the VM OS, we only observed loc interrupts triggered at a rate of 250 Hz.

Despite our efforts to lower the number of interrupts by applying DPDK, there still remain a number of interrupts triggered by the OS itself, causing latency spikes. Due to their scarcity and limited duration, we do not consider them harmful to our pursuit of building a latency-optimized system considering the URLLC latency goals.

Influence of batch sizes

All previous measurements use CBR traffic. For CBR, the pauses in between packets can be used for packet processing without delaying subsequent packets leading to optimal latency results.

However, real traffic may arrive in bursts of packets without pauses between them. There, packet processing may delay subsequent packets. The following measurements show the impact of bursty traffic on the latency.

We define a block of packets arriving back-to-back on the wire as a burst and a block of packets being accepted or processed on a device as batch. Batched packet processing leads to higher throughput for packet processing frameworks like DPDK (cf. Section 5.1.5).The DPDK-enabled Snort accepts batches of up to 32 packets, processes them, and then releases the batch of packets only after all batched packets have been processed. Figure 5.14a shows the results of this processing strategy for different batch sizes. All graphs show areas of very steep increases indicating a large number of packets sharing the same latency, i.e., a batch of packets is sent out. Starting with a batch size of 4, flat areas become visible, indicating that no packets were observed with this latency, i.e., the batch is processed without any packet sent out. The flat areas are followed by steep increases where the batches are sent out and the flat areas grow with increasing batch sizes as batch processing times increase. For a 64-packet burst, a two-step pattern is observed as two batches of 32 packets are processed in sequence. The plots show few packets with lower latency for every burst size. This happens if only a few packets of a burst are put into a batch, processed, and sent out before the remaining packets of the burst are processed.

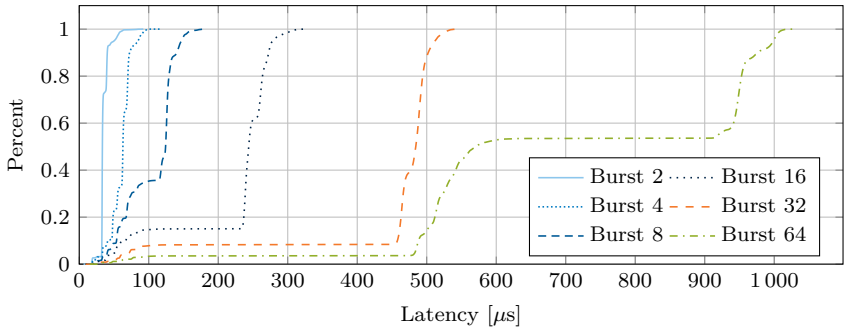
As already processed packets are delayed until the batch is fully processed, the median delay is raised significantly. For low-latency optimized systems, smaller batch sizes can be beneficial. Therefore, we change the batch size from 32 to the minimal DPDK-supported batch size of 4. The results can be seen in Figure 5.14b, where the CDFs display a linear trend for growing burst sizes. This distribution results in a significantly lower median for burst sizes of 16 and above with little influence on the maximum observed latency.

We have shown that the blocking behavior of this batch-processing strategy may increase latency unnecessarily. For low-latency systems, small batch sizes or even no-batch processing decrease latency. However, large bursts may cause latency violations due to short-time overload scenarios. In our case, burst sizes of 32 and 64 lead to latencies not meeting the URLLC criteria any longer for the chosen scenario.

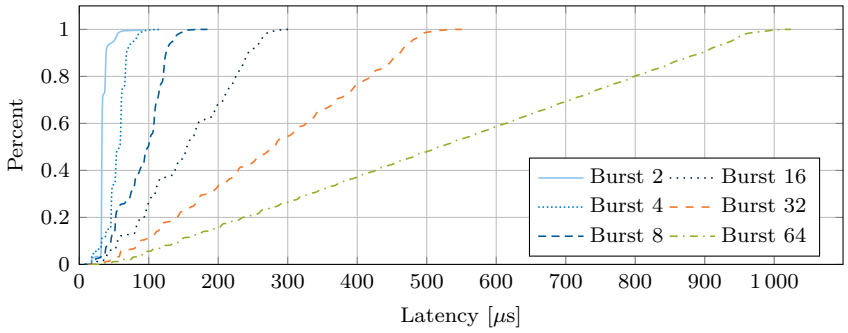
Energy consumption

Our proposed low-latency configuration requires deactivating energy-saving mechanisms. Therefore, we compare the system configuration used for testing with a configuration with default BIOS settings and kernel arguments for energy saving enabled. For the power measurement, we use the metered power outlet Gude Expert Power Control 8226-1. We measure the power consumption of the entire server.

Table 5.4 lists the measured power values. We observe no differences in power consumption between the different applications (DPDK-l2fwd, Snort-fwd, Snort-filter). We measure the server while idling, while the application is in an available state, and while the application is actively processing packets. With power saving enabled, there is a 14-watt difference between idle and transmitting state and a 3-watt difference between running and transmitting. The latter, rather



(a) 32-batch processing



(b) 4-batch processing

Figure 5.14: Latency when forwarding using Snort-filter (VM) at 10 kpkts/s for different burst sizes (cf. Gallenmüller et al. [4])

Power saving	Idle	Available	Processing
enabled	31 W	42 W	45 W
disabled	46 W	47 W	47 W

Table 5.4: Power consumption (cf. Gallenmüller et al. [4])

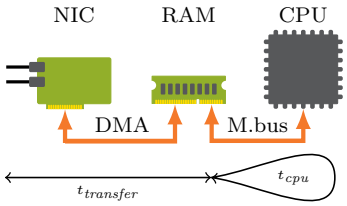


Figure 5.15: Sources of delay on modern architectures (cf. Gallenmüller et al. [4])

low difference, is a consequence of DPDK's design, relying on active polling, therefore, keeping the system (re-)active even without packet transfer. This intentional design decision of DPDK makes it a well-suited framework for high-performance scenarios where energy consumption is always high. However, DPDK is a poor choice for scenarios with low load because of the high energy consumption.

Disabling power saving increases the previous maximum power consumption by 1 W for the idle state and 2 W for the other states. Comparing power saving enabled and disabled shows that low-latency configuration does not come for free. In scenarios with long idling periods, power consumption and costs rise by 48 %. For other load scenarios, the increase is lower (12 and 4 %), i.e., if the system load is already high for the traditional systems, the additional costs for the low-latency configuration are significantly lower. The CPU used in our test system has a thermal design power (TDP) of 35 W, running more powerful CPUs with energy saving disabled may introduce even higher differences between idle and running states and, therefore, higher costs.

5.4.4 Model

Our measurements have shown that the packet processing system must not be overloaded to adhere to URLLC requirements. Therefore, we deduce a model calculating the maximum packet rate our system can handle without overloading.

Figure 5.15 shows the path of a packet through the different system components and the associated time consumptions t. The time a packet travels through the system ($t_{transfer}$) includes delays caused by propagation, serialization, and the transfer from NIC to RAM. t_{CPU} denotes the time the CPU processes the packet. As packets are received and sent, the path of a packet involves $t_{transfer}$ twice, assuming symmetrical receiving and sending delays. This assumption leads to Equation 5.9 for calculating the end-to-end delay of a single packet.

$$t_{e2e} = t_{CPU} + 2t_{transfer} \tag{5.9}$$

		DPDK-l2fwd $2t_{transfer}$ [μs]	Med. latency t_{e2e} [μs]	CPUtime t_{CPU} [μs]	Max. Rate R_{max} [kpkts/s]
Snort-fwd	HW	3.1	14.5	11.4	87.4
	VM	3.3	15.9	12.6	78.7
Snort-filter	HW	3.1	17.4	14.3	69.7
	VM	3.3	18.4	15.1	65.6

Table 5.5: Calculated CPU times and maximum rate (cf. Gallenmüller et al. [4])

	loc_{host}		iwi_{host}		loc_{VM}		d_Σ per s
	r [Hz]	d [μs]	r [Hz]	d [μs]	r [Hz]	d [μs]	[μs]
HW	166.7	10.9	83.3	13.6	-	-	2949.9
VM	166.7	17.5	83.3	19.2	250	17.5	8891.6

Table 5.6: Trigger rates (r) & delays (d) of interrupts (cf. Gallenmüller et al. [4])

Section 5.1 identifies the CPU as one of the main bottlenecks in software packet processing. Especially considering the low packet rates (below 120 kpkts/s), neither the Ethernet bandwidth, the NIC, nor the involved system buses are overloaded. Subsequently, it is crucial to determine the required calculation time on the CPU, t_{CPU}, for calculating the maximum packet rate. Table 5.3 lists the measured end-to-end delays of the packets (t_{e2e}) not t_{CPU}. However, the DPDK-l2fwd scenario—representing the most basic forwarder possible without any processing for the packet—involves only a minimal amount of t_{CPU}.

Measurements in Section 5.1 have shown that DPDK uses 100 CPU cycles for receiving and transmitting a packet (c_{IO}). On a CPU with a clock frequency of 2.2 GHz, 100 cycles result in a delay of 45 ns. We measured a median end-to-end delay for DPDK of 3.1 μs, which makes the impact of the IO operation on the CPU negligible.

Therefore, we can use the median value of the DPDK-l2fwd measurement as an approximation of $2t_{transfer}$. Using that information, we can calculate an approximation of t_{CPU} for Snort-fwd and Snort-filter, by deducting the median measured in the DPDK-l2fwd scenario from their respective end-to-end delays. The results of the approximated t_{CPU} are given in Table 5.5.

Section 5.4.3 shows that a CPU core also performs interrupts. Table 5.6 lists CPU time spent on interrupts per second d_Σ depending on the scenario, the interrupt rates, and the costs of the individual interrupts. Knowing the amount of CPU time spent on packet processing per packet (t_{CPU}) and d_Σ, Equation 5.10 can be deduced. The maximum packet rates calculated according to this equation are listed in Table 5.5.

$$R_{max} = \frac{1\,\text{s} - d_\Sigma}{t_{CPU}} \tag{5.10}$$

Comparing the calculated maximum rates in Table 5.5 with the actual maximum rates measured in Table 5.3 shows that Equation 5.10 can predict the overload correctly for three out of four scenarios. For the Snort-fwd (VM) scenario, the maximum rate is underestimated with the

overload not happening at the predicted rate 78.7 kpkts/s but beyond 80 kpkts/s. We conclude that the prediction approximates a lower bound for the maximum packet rate. A conservative approximation is advisable in this scenario, especially considering the devastating impact of overload on latency and QoS.

5.4.5 Limitations

Despite its benefits in terms of latency and jitter, the proposed architecture has disadvantages. Statically assigning VMs to cores does not allow sharing a CPU core between several VMs, at least not the isolated cores dedicated to realtime applications. This core allocation strategy increases hosting costs for such a VM. Migrating VMs or scaling the VM setup is not possible as SR-IOV does not allow VM migration due to the non-trivial replication of the NIC's hardware state [98]. Disabling energy-saving mechanisms increases energy costs for the server (cf. Section 5.4.3), air conditioning, and increases the thermal load on the hardware, which in turn may require earlier replacement additionally raising costs.

5.4.6 Reproducibility

As part of our ongoing effort towards reproducible network research, we release the pcap traces and plotting tools used in the measurements of Section 5.4. Further, we release our measurement tools and source code of the investigated software, including a detailed description for others to replicate our measurements as a GitHub repository [106]. Table 5.3 and Figures 5.10, 5.13, and 5.14 are explained in detail, i.e., the used source code, experiment scripts, generated data, and plotting tools.

5.4.7 Conclusion

Our analysis shows that—in contrast to non-optimized systems—a carefully tuned system architecture meets the demanding latency and reliability requirements of future 5G URLLC services. Hardware-timestamped latency measurements of the entire network traffic, allow for a detailed analysis of worst-case latencies, bursty traffic, and system load. We measured a virtualized system running a real-world intrusion prevention system causing a worst-case latency of 116 μs on a steady-state system, leaving enough room for subsequent packet processing tasks. Further, we show bursty traffic causing short-time overloads violating the latency requirements and introduce a strategy to reduce its impact. By publicly releasing our experiment scripts and data, we provide the foundation for others to reproduce all measurements described in this thesis.

We introduce a model to predict system overload to avoid the destructive effect of overload on latency. The benefits of this model are its simplicity requiring only the median forwarding latency for IO and the interrupt processing times.

Despite the increase in power consumption (48 % for a low-load and 4 % for a high-load scenario), we demonstrate that off-the-shelf hardware and available open-source software can achieve consistently low latency. Relying on established hardware and tools simplifies the transition towards URLLC.

For future work, we want to investigate the impact of hosting different 5G service classes on the same system, especially regarding potential QoS cross-talk and potential mitigation strategies.

5.5 Key Results

This chapter presents a series of measurements for high-performance packet processing applications. These applications are based on specialized frameworks for userspace packet processing that are optimized for high throughput. We investigate three different frameworks, netmap, PF_RING ZC, and DPDK. Our measurements demonstrate that DPDK offers the best performance considering throughput and latency. Besides, we investigate Snabb, a packet processing framework with a focus on NFC applications. We show that Snabb can provide similar performance to DPDK applications. Further, we present MoonRoute, a DPDK-based software router that demonstrates a highly scalable software application that can utilize modern multi-core and many-core architectures. A low-latency software architecture is created and investigated using a DPDK-enabled intrusion prevention system.

We show that realistic performance emulation of packet processing tasks is possible without an actual implementation of this task. Therefore, we introduce the tool SHEEP that can emulate arbitrary CPU and cache load. With the correct parameters, a SHEEP-enabled NF can recreate the impact of arbitrary complex packet processing operations on performance.

All presented measurements are described using the previously introduced resource model. The resource model is applied to predict the throughput performance and scalability of various packet processing tasks—packet forwarding, NFC, and routing. Moreover, the resource model is used to predict system overload to avoid the impact of overload on the latency of a packet processing system.

5.6 Author's Contributions

Section 5.1 is based on joint work by Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle [38]. The author created the measurements, their analysis, and an early version of the model for his Master's Thesis [39]. For the paper, the analysis part was recreated and latency measurements were added. For this thesis, the model was improved and refined.

Section 5.2 presents a measurement methodology published by Wolfgang Hahn, Borislava Gajic, Florian Wohlfart, Daniel Raumer, Paul Emmerich, Sebastian Gallenmüller, and Georg Carle [78]. The author contributed significantly to the development and measurements of the emulated NF, based on the previously introduced SHEEP [77] framework. This thesis applies the resource model to the data presented in the original paper and gives a detailed analysis of the impact of caching on NFC performance.

Section 5.3 is based on two joint publications between Sebastian Gallenmüller, Paul Emmerich, Rainer Schönberger, Daniel Raumer, and Georg Carle [80] and Paul Emmerich, Sebastian Gallenmüller, Rainer Schönberger, Daniel Raumer, and Georg Carle [81]. Design and measurements

are based on a Master's Thesis by Rainer Schönberger, supervised by the other co-authors. The author contributed the description of MoonRoute, the measurement graphs, and the analysis for this publication.

Section 5.4 is joint work between Sebastian Gallenmüller, Johannes Naab, Iris Adam, and Georg Carle [4]. The author identified the required system configurations, performed and released the measurements, conducted their analysis, and derived the model presented.

Chapter 6

Measuring and Modeling of Networked Control Systems

This chapter applies a data-driven measurement methodology to the domain of networked control systems (NCS). For this application domain, there exists no common benchmarking methodology like the RFC 2544 for fixed networks. Therefore, we develop our own benchmarking suite, called NCSbench, dedicated to the application domain of NCS. NCSbench consists of the benchmark methodology and a reference platform for NCS, including hardware and software. Measurements are presented to evaluate the repeatability and replicability of NCSbench.

In a second step, we transfer the ideas and concepts of our fixed-network testbeds to wireless networks. This testbed demonstrates a possible way towards reproducible network experiments for IEEE 802.11 WLAN networks.

6.1 Benchmarking Networked Control Systems

Section 6.1 is based on joint publications by Sebastian Gallenmüller, Stephan Günther, Maurice Leclaire, Samuele Zoppi, Fabio Molinari, Richard Schöffauer, Wolfgang Kellerer, and Georg Carle [107] and between Samuele Zoppi, Onur Ayan, Fabio Molinari, Zenit Music, Sebastian Gallenmüller, Georg Carle, and Wolfgang Kellerer [108].

A cyber-physical system (CPS) can be divided into two components: the actual physical system or plant, and a microcontroller managing this plant. If the microcontroller controls a process on the plant, both components form a control system. Traditionally, controller and plant are integrated into the same device, forming a robust, self-contained control system. This thesis investigates a type of CPS where controller and plant are separate devices exchanging information through a network creating an NCS [109]. NCS are common in industrial applications, e.g., in the closed-loop regulatory control of production machines [110]. Due to their importance, NCS are widely discussed and modeled under different operating conditions [111].

Despite a large number of results achieved by the control and networking research communities, the reproducibility and comparison of experimental NCS results are still obstacles to overcome. These shortcomings are challenging for different reasons. An NCS requires the expertise of formerly separated domains, namely the control domain and the network domain. Both disciplines have established their own procedures and methodologies for comparing and rating their systems' performance. We aim for a unified approach for benchmarking NCS: we describe a common methodology for benchmarking the control as well as the network aspects of an NCS. We specify a common benchmark scenario defining the relevant KPIs to determine the quality of the control process and the network through repeatable experiments. In addition, we describe the application of this benchmark on an example platform. To that end, we develop an NCS based on the widely used Lego Mindstorms, which we want to evaluate using our benchmark. Construction manuals and software are available as open source to establish a low-cost benchmarking framework for NCS, which can be replicated easily by other researchers.

Our benchmarking suite is called NCSbench [112]. NCSbench,

1. proposes a novel NCS benchmarking methodology based on the joint expertise of control and network domain,
2. presents the implementation details of the first open-source NCS benchmarking platform designed for replicable results, and
3. evaluates the replicability of the platform and the validity of the methodology with experiments in different scenarios.

Section 6.1.1 lists related work on the topic of NCS benchmarking. In Section 6.1.2, we introduce our benchmark before presenting the KPIs for network and control domain in Sections 6.1.3 and 6.1.4. Sections 6.1.5 and 6.1.6 explain the benchmarking methodology and the architecture of the benchmarking framework. We model the delay in Section 6.1.7.

6.1.1 Related Work

The problem of modeling the effects and constraints introduced by networks on control systems is well studied in literature [111], [113], [114]. Zhang et al. [111] study the constraints of the network on control systems, together with practical applications arising from NCS. A traditional, control-oriented approach to the problem is to model the network as a source of random delays and dropouts [113]. Despite being well studied, there are still challenges in modeling the influence of the network on the control system, e.g., distributed controllers [114].

Lu et al. [115] state that conveying full-scale practical research with a real implementation of a CPS is a difficult task due to the complexity and the replicability of experimental platforms. Therefore, research work in the field of NCS conducting experimental studies is limited. Zhang et al. [116] and Chamaken et al. [117] implement a hybrid setup of an NCS combining hardware-in-the-loop, i.e., a simulation of the plant dynamics, with a real network. Kawka et al. [118] and Eker et al. [119] use the network-in-the-loop approach, i.e., a simulated network, with real hardware as a control system. A different research approach provides prominent examples where the complete NCS consists of real hardware [120]–[124]. Bachhuber et al. [120]

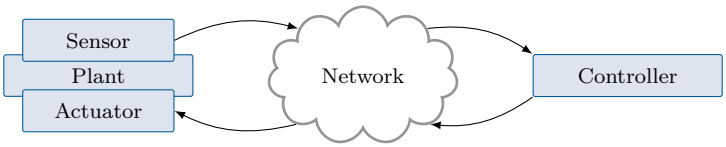


Figure 6.1: Control loop of an NCS (cf. Gallenmüller et al. [107])

conduct an end-to-end latency analysis of a vision-based NCS. Baumann et al. [121] present measurement results from the case study of balancing an inverted pendulum over a multi-hop wireless network. Mager et al. [122] propose a reliable multi-hop wireless protocol that enables the remote control of multiple experimental inverted pendulums. Ploplys et al. [123] use a rotating inverted pendulum controlled via WLAN. Peng et al. [124] demonstrate an inverted pendulum utilizing low-power wireless networks. All these implementations perform real-world measurements of standard NCS to prove the validity of their contributions. However, they use specific hardware and software solutions, which, together with different and scattered measurement scenarios, introduce a significant obstacle for reproducibility. For this reason, different techniques are difficult to compare in a single, repeatable NCS benchmark scenario.

Similarly, there exist standard benchmarks that focus exclusively on the network domain, such as RFC 2544 [1]. Network-focused benchmarks may provide only limited insight into the behavior of specialized applications, such as NCS.

Although practical NCS implementations pose a major challenge for reproducing NCS experiments, conceptual CPS benchmarking scenarios have been defined in literature [125]–[129]. Nethi et al. [125] present a platform for the emulation of NCS to enable their comparison in different scenarios. Wu et al. [126] develop FARE, a framework for benchmarking the reliability of CPS, not tackling, however, the specific aspects of NCS. Ding et al. [127] propose a framework for the design of fault-tolerant industrial NCS, parametrizing the network and the control systems. The framework allows experiments with real NCS, but does not tackle the aspects of reproducibility and comparison. Boano et al. [128] elaborate on how to implement experimental benchmarks and define KPIs to compare experimental results. Nonetheless, they do not conduct a practical study to verify the proposed methods in their own work. Niemueller et al. [129] present a definition of the elements needed to enable the benchmarking of different NCS. Here, the KPIs and context of an industrial CPS are identified and presented in a holistic benchmark scenario. However, this benchmark is limited to high-level multi-robot systems and does not cover the low-level interconnections of closed-loop NCS. To the best of our knowledge, none of the existing literature tackles the problem of reproducibility and benchmarking in a full-scale practical scenario. Therefore, we present the first replicable, experimental platform and practical benchmarking methodology for NCS.

6.1.2 Framework for Reproducible NCS Benchmarking

A simple model of an NCS is shown in Figure 6.1. This control system involves a single controller and a single plant visualizing the typical flow of information in a control loop—additional information flows, e.g., the initial configuration of the sensor, are neglected. The sensor and

the actuator of the plant are connected to the controller over an arbitrary wired or wireless network. Throughout this thesis, we reduce the investigated systems to the minimum number of components involved. Here, we opted for a two-node setup, which is still powerful enough to demonstrate our benchmarking framework while minimizing the chance of misconfiguration and simplifying the reproduction of experiments.

Benchmarking framework: Our framework defines a *benchmark* as a series of repeatable experiments. In each experiment, a set of values is measured. In the context of our benchmark, an *experiment* is a time-bounded execution of the NCS platform, measuring a real-world setup. Alternatively, values can be obtained through simulation or emulation. A whole series of different experiments may be necessary to achieve the expected behavior of the NCS. We call the process of identifying these conditions *challenge*. The results of the experiments depend on the conditions of the experiment during the time of execution. An entire set of conditions relevant to an experiment defines a *scenario*, which includes the set of parameters relevant for evaluation. To make the experiments and ultimately the benchmarks repeatable, replicable, and reproducible, all relevant information of a scenario must be documented in a *scenario description* specifying the:

1. *application software* with a specific controller or plant,
2. *network stack* including the various protocols and technologies,
3. *network topology* describing the connectivity between the nodes of the network,
4. *physical environment* such as distance of nodes, model, and parameters of the channel model and noise floor (signal-to-noise ratio, SNR),
5. *interference* with other nodes in the network or with external transmitters outside the network (signal-to-interference-plus-noise ratio, SINR), and
6. *hardware* of the controller, the network, and the plant.

An example of such a scenario description is given in Table 6.1. The outcome of an experiment is a set of measured values and parameters. Utilizing these results, we derive *key performance indicators* describing the quality of the whole system or subcomponents efficiently and reproducibly. As the CPS and the network require their own KPIs, we decided to split these KPIs along the layers defined by the ISO/OSI model with the network KPIs on Layers 4 to 1 (see Section 6.1.3) and the control system KPIs on Layer 7 (see Section 6.1.4). Figure 6.2 depicts the vantage points of our measurements on the left side. The layers are investigated separately due to different network behavior. We consider our benchmark to be network-agnostic, i.e., we want to be able to apply it using different network technologies such as wireless LAN (IEEE 802.11), Ethernet (IEEE 802.3), or others, e.g., low power wireless networks (IEEE 802.15.4). Therefore, we specify only the interface to the highest layer we want to investigate. For instance, we specify using UDP when investigating the network starting at the transport layer.

Challenge: The challenge is central to our benchmarking framework to identify scenarios where the NCS can operate successfully. When designing an NCS, one usually expects a certain service

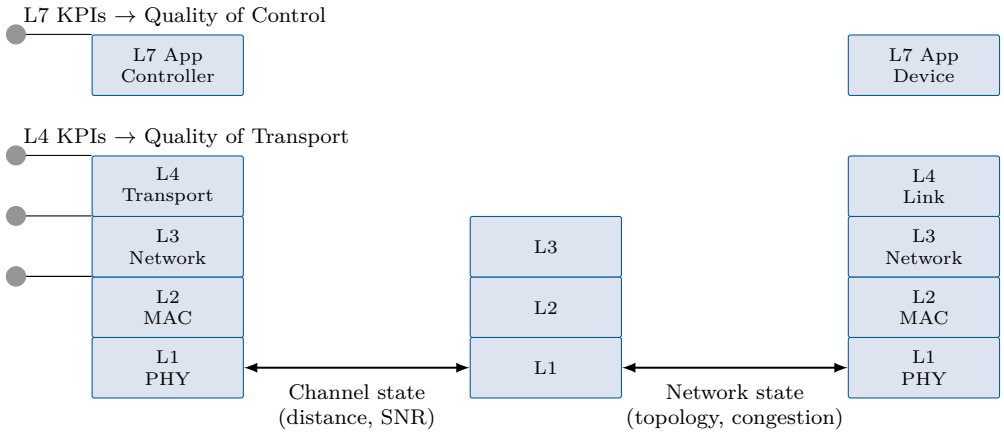


Figure 6.2: KPIs on different layers of the ISO/OSI stack (cf. Gallenmüller et al. [107])

level from the whole system, i.e., the quality of control (QoC). This QoC can be expressed using Layer 7 KPIs. Our benchmarking framework aims to find scenarios, i.e., the conditions that can fulfill these requirements. Based on specific, desired KPIs, it may be possible to directly infer the required quality of the network KPIs, e.g., the maximum allowable delay. Directly inferring the required quality reduces the set of possible scenarios for testing. In other cases, the network KPIs may not be directly inferable from the Layer 7 KPIs. For these cases, scenarios can only be determined experimentally by gradually modifying the conditions of the scenarios.

In the following, Section 6.1.3 describes KPIs to characterize network links, such as packet rate, loss rate, delay, inter-packet time, jitter, and bandwidth-delay product. Section 6.1.4 focuses on the control system, presenting two different sets of KPIs: a set of generic KPIs applicable to a wide range of different control systems and set of specific KPIs adapted to our specific DuT—a two-wheeled inverted pendulum.

6.1.3 Network Domain KPIs

The following paragraphs discuss network KPIs with a particular focus on the requirements of NCS.

Packet rate: The packet rate denotes the number of packets being transmitted or received. Typically, per-packet costs dominate over the per-byte costs making the packet rate more critical than the actual throughput in Gbit/s (cf. Section 6.4). Since the maximum packet rate is limited by per-packet overheads, such as processing time and medium access, it is often more important than the actual data rate, which additionally depends on the packet size.

The average rate at which packets are transmitted at the source node may differ from the rate packets are received due to losses. When referring to the rate at the destination, the term *goodput* may be used to describe the gross data rate minus actual losses in the network. The rate may differ depending on the layer of the ISO/OSI model that is being considered: multiple messages transmitted by a controller may be aggregated at the transport layer and sent as a

single frame on the link layer. Conversely, a large message may be split into multiple packets at the network layer and transmitted as multiple individual frames while being reassembled at the destination's network layer. Therefore, it is essential to state which layer one is referring to.

Loss rate: The loss rate denotes the fraction of packets transmitted but not received. Given the number of packets p that were successfully received and the number q of missed packets, the loss rate is given as $\epsilon = q/(p+q)$. Besides the mere rate, the pattern in which the losses occur can be described.

In practice, the receiver's loss rate can most easily be determined using sequence numbers in each transmitted packet. Sequence numbers can be used on different layers, e.g., Layer 2 for unreliable wireless networks, Layer 4 protocols like TCP, or even the application layer. The transmitter inserts the sequence numbers. When a receiver detects a gap in the sequence number of subsequent packets, it can determine the exact number of missed packets in between. Note that this demands that no reordering occurs after sequence numbers were chosen.

Delay: The term *delay* commonly refers to the total time needed to transmit (medium access plus serialization) a packet and forward it to the destination (propagation delay plus processing and buffering delays at intermediate nodes). It is often referred to as *one-way delay* to avoid confusion with the *round trip time (RTT)*. For non-trivial networks, where packets can take several routes through the network, the delays should be specified for both directions separately, as delays may differ significantly.

The time for medium access significantly differs depending on the actual implementation of the medium access and physical layer: while it is in the single-digit microsecond range and rather constant for switched, full-duplex Gigabit Ethernet networks [74], it is orders of magnitude larger in wireless networks due to the complex medium access strategy and the shared nature of the medium [130]. If multiple transmitters contend for medium access, both the delay and its standard deviation may well be in the range of milliseconds for individual nodes.

Timestamps can be acquired from the hardware of network interfaces for transmitted or received packets. To timestamp events in software, clock counters of CPUs can be used, such as the TSC on x86 CPUs. For synchronizing timestamps across different devices, clocks need to be synchronized, for instance, by utilizing protocols such as PTP. These protocols offer higher accuracy if the synchronization is done via a wired connection. Therefore, an additional wired connection beside a wireless connection is beneficial during the execution of the benchmark.

The serialization delay can be approximated by the frame size L and the bitrate R once control over the medium is gained, i.e., $d_s = L/R$. WLAN [131] prepends signaling information on the physical layer at a different data rate during the physical layer convergence procedure. Therefore, the relation between frame length and bitrate is only an approximation. The impact of the serialization delay on the overall delay decreases for growing bandwidths. For bandwidths of multiple Gbit/s, serialization delay is in the order of nanoseconds. In NCS, where the delay is measured in the order of milliseconds, the influence of the serialization delay can be neglected.

The propagation delay depends on the distance s a signal has to travel, the speed of light c, and a medium-specific constant ν: $d_p = s/\nu c$. The constant ν is approximately one for wireless transmissions in air and vacuum, roughly 2/3 for copper cables, and slightly larger in fibers [16]. For transmission in local networks, the propagation delay can be neglected but may be the dominating part of long-range or satellite transmissions.

Finally, processing and buffering delays differ depending on the nodes along the network path from source to destination and the current load of the individual nodes. In general, those delays are challenging to quantize and particularly hard to measure without direct access to the respective node.

Whether or not individual summands of the delay may be neglected depends on the demands of the NCS. For our example of the inverted pendulum, we investigated the delay caused by different subcomponents of the control process (cf. Section 6.1.7).

Inter-packet time & jitter: The *inter-packet time (IPT)* is defined as the time distance between two subsequent packets in a packet stream. Sensor values of control systems are typically read at a fixed rate, which leads to a constant inter-packet time for NCS. *Jitter* is defined as the difference between the IPTs at the source and destination for two subsequent packets. In traditional, tightly integrated control systems, jitter was negligible. This stable environment led to the design of controllers, which could rely on a constant flow of precisely timed sensor data. For NCS, the inter-packet time can have a significant impact, especially if running a controller, which expects sensor data to arrive at a constant delay [132]. Therefore, our benchmark considers the jitter to be a valuable KPI for controller design.

Bandwidth-delay product: The *bandwidth-delay product* commonly expresses the amount of data in flight between source and destination, and is thus expressed in bit. However, it may also be used to quantify the number of packets currently in flight between source and destination. For the scope of an NCS, the latter is more relevant as packets containing sensor data or control feedback are typically small with a constant size. Furthermore, sensor data that arrives too late at the controller is commonly considered as loss, which is why the aggregation of multiple sensor values into single packets is expected to be of little help. In the following, we express the bandwidth-delay product as a number of packets concurrently in flight from source to destination.

6.1.4 Control Domain KPIs

We design our KPIs to measure the robustness and the performance of a control process. Robustness is the ability of the control system to counteract disturbances. KPIs measuring the robustness quantify the effectiveness of a system to revert to its reference state in the presence of disturbance. The performance of a control system describes its efficiency, e.g., the time a control process needs to revert to the reference state.

We propose KPIs classified into two groups—*generic* and *specific KPIs*. Generic KPIs are designed to measure arbitrary control systems; specific KPIs are chosen to describe a particular

type of control system. Specific KPIs allow extending our benchmark to meet the requirements of different control systems. For NCSbench, we define specific KPIs to measure our network-controlled two-wheeled inverted pendulum.

Generic KPIs: We consider control systems that steer a system to an individual reference state, e.g., a two-wheeled inverted pendulum robot (TWIPR) that is trying to maintain an upright position. Therefore, we define three generic KPIs:

- The maximum disturbance that may occur such that the state can still be steered back to the reference.
- The time it takes to steer the state back to reference after applying a specific disturbance.
- The energy needed to steer the system back to reference after a specific disturbance. This energy can be measured and integrated over the entire measurement period.

Specific KPIs: Concerning the example of the inverted pendulum control problem, we define KPIs specific to the inverted pendulum or similar systems. Further, we assume that the inverted pendulum starts in its upright position, its reference state. We define the following KPIs:

- Difference of the reference position and the actual position of the inverted pendulum, which can be expressed as an angle between the current and the upright position of the pendulum.
- Tracking of the wheel movement, which can be measured as the angle of the current motor position with respect to its initial reference state.

Both specific KPIs can be integrated over the measurement period. An ideal controller could control the pendulum with a minimal number of actions, correcting only a small value, which would result in low values for both KPIs. If the number of actions or the value of the actions increases, so do the KPIs, indicating a decrease in the QoC.

Reproducibility vs. Non-Determinism

Certain KPIs can behave non-deterministically, such as the delay or packet loss. Reporting only an aggregated number, such as average or median, as a KPI does not suffice to enable repeatable experiments. Therefore, we propose to additionally report more descriptive data such as entire logs or histograms, allowing to model repeatable behavior. Disturbances or interferences are essential features to explain the CPS KPIs. These should also be quantified and reported accordingly to aid the process of understanding and repeating the observed behavior of the CPS.

6.1.5 Evaluation Platform

Our goal for the evaluation platform is a simple, low-cost platform that can easily be extended to allow others to replicate our setup and experiments. We opted for a well-known NCS setup: an inverted pendulum as shown in Figure 6.3. The inverted pendulum is built from Lego Mindstorms, which is widely available, reasonably priced, and easily extensible either in software or hardware.

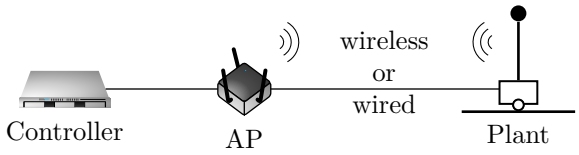


Figure 6.3: Two-hop network topology used in NCSbench, supporting Ethernet and WLAN USB adapters (cf. Gallenmüller et al. [107])

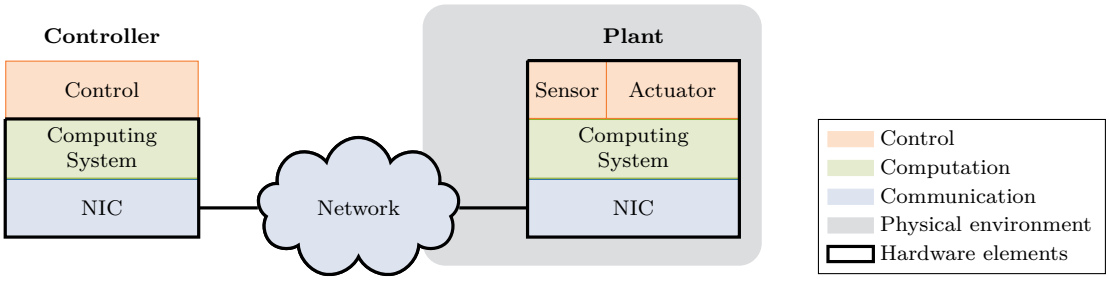


Figure 6.4: Architecture of the NCS platform (cf. Zoppi et al. [108])

Our benchmark methodology relies on the combined knowledge of control, computation, and communication domains and the experience gained during the implementation of the proposed NCS platform. We do not only extend the existing methodologies [128], including experimental knowledge, but provide a novel approach to model the architectural elements (Section 6.1.6) and the delays of NCS (Section 6.1.7).

The purpose of the benchmarking methodology is to define the necessary amount of information to reproduce and evaluate experimental results using the NCS platform. Following ACM's reproducibility terminology [5], we first want to recreate our own results, thereby establishing repeatability. In a second step, we recreate the NCS benchmark across the different involved research groups making our results replicable. We provide the entire framework containing the source code, the plotting scripts, and the measurements of our framework as open source, thereby encouraging others to recreate our results and fostering the development towards a fully reproducible benchmark.

6.1.6 NCS Architecture and Scenario Description

We propose an architecture for experimental NCS as depicted in Figure 6.4. The architecture is composed of several software and hardware elements organized according to the three CPS domains [133]: control, computation, and communication.

The set of elements composing the *control system* is twofold. On one side, the plant, i.e., the robot, mounts sensors and actuators capable of sensing the physical system and executing the actuation commands. On the other side, the controller, detached from the plant, receives the sensor readings, executes the control logic, and transmits instructions to the actuator.

Two different *computing systems* provide computing power and access to the network interfaces to both controller and plant. The interconnection of the control application with the network interface is achieved by implementing the upper-layer protocols of the OSI communication stack.

The *communication network* physically interconnects the computing system of the controller with the computing system of the plant and enables the flow of information between them. In our architecture, it defines the lower layers of the OSI communication stack.

To make the experiments and ultimately the benchmarks reproducible, it is vital to document all relevant information of the NCS architecture in a *scenario description*. For every component of the NCS architecture of Figure 6.4, software (algorithms) and hardware parameters must be specified to replicate the experiments.

Control Parameters

The *control application software* runs on the computing systems and implements the control logic that drives the NCS. The *physical system* describes the physical properties of the robot itself. Further, we list the *hardware* used on the robot, such as the used sensors and actuators.

Network Parameters

The *network topology* describes the connectivity between the nodes of the network.

In the scenario description, all the network parameters are part of the *lower layers*. These lower layers involve all functions that are part of the network layer, link layer, and physical layer, which are implemented in the *network stack* and *network drivers* of the OS, and in the *firmware* executed by the NICs.

The *network hardware* is part of the network parameters, listing the hardware models of the network interfaces used by robot and controller.

The *physical environment* defines the physical conditions that the network operates in, such as the interference with other wireless nodes. These properties strongly affect wireless networks, making them inherently difficult to reproduce without a radiofrequency shielded test environment. For our benchmark, we try to minimize the impact of the physical environment on the measured results. Our benchmark should be widely replicable across different research groups. Therefore, we decided not to require access to a shielded test environment. For this reason, we suggest executing the benchmark in an environment with low wireless network activity, thereby minimizing the impact of external interference and moving objects on the measurement results. For benchmarks using wired networks, such as a full-duplex switched Ethernet, the physical environment has no impact on the measurement results as long as the network is not overloaded. Therefore, wired network measurements are easier to reproduce and can even be used to emulate the behavior of wireless networks on the network layer.

Computing System Parameters

The *higher layers*, i.e., the transport layer and higher layer protocols, are part of the computing system, connecting the control and the networking domains of the NCS. The *transport protocol* is implemented in the OS; therefore, the OS version is required for describing the computing

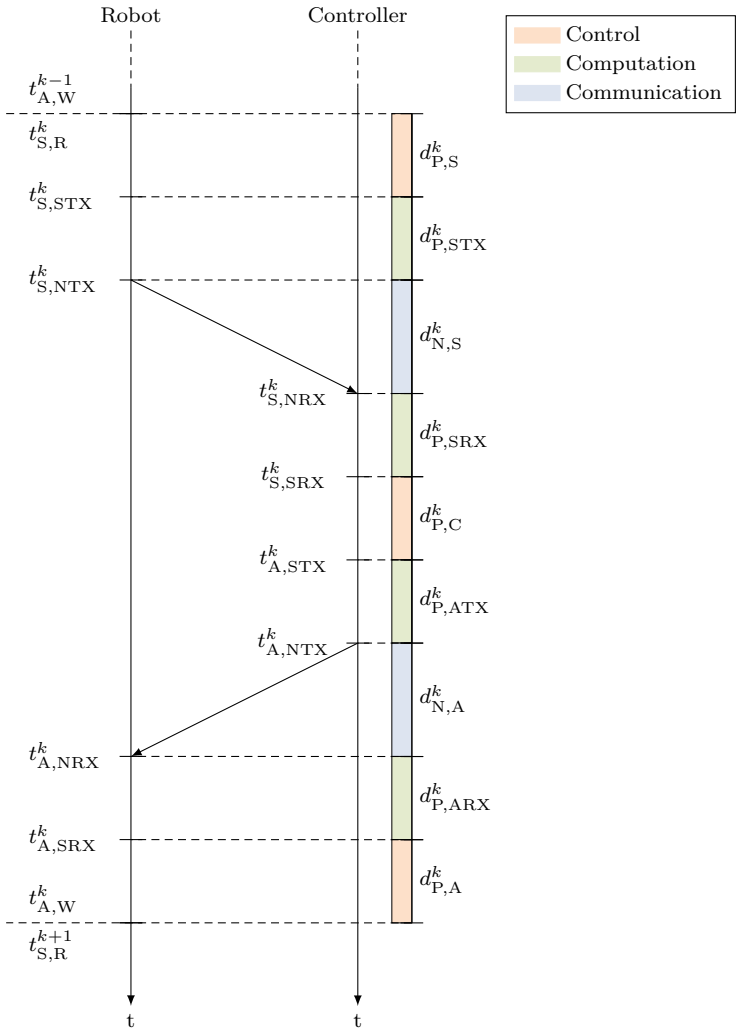


Figure 6.5: Model of the timings of an NCS together with the *processing* (P) and *networking* (N) delays of the control, computation, and communication CPS domains (cf. Zoppi et al. [108])

system parameters. The *application protocol* is required for the logical exchange of sensor values and actuation commands between the controller and the plant.

The computing systems utilize *hardware*, which provides computing power and access to the communication facilities.

6.1.7 Timings and Delay Model

When all the components of the NCS architecture are interconnected, the information regularly flows between the plant and the controller over the communication network. In particular, every sampling period, the sensor measures the state of a plant and sends it to the controller, which computes and sends a command to the actuator that steers the plant.

The time evolution of the k-th sampling period is shown in Figure 6.5. At time $t^k_{\mathrm{S,R}}$, the sensor values (S) of the plant's sensors are read (R), handed over to the plant's network stack (STX) at $t^k_{\mathrm{S,STX}}$ and transmitted over the communication network (NTX) at $t^k_{\mathrm{S,NTX}}$. The controller's network interface receives the sensor data (NRX) at $t^k_{\mathrm{S,NRX}}$ and its network stack delivers the packet to the control application (SRX) at time $t^k_{\mathrm{S,SRX}}$. Afterward, the controller calculates the actuation values for the actuators and hands over the actuation message (A) to the network stack at $t^k_{\mathrm{A,STX}}$, which sends the packet over the network at $t^k_{\mathrm{A,NTX}}$. Finally, at time $t^k_{\mathrm{A,NRX}}$, the plant's network interface receives the actuation packet, and, at time $t^k_{\mathrm{A,SRX}}$, its network stack delivers it to the actuator application, which applies (W) the commands to the actuators at $t^k_{\mathrm{A,W}}$.

Thanks to the timing diagram shown in Figure 6.5, it is possible to identify the delay components of the NCS and distinguish the delays arising from the control system, computing system, and communication network. Control system delays arise from the *processing time* (P) of the control algorithms. At the robot during sensing $d^k_{\mathrm{P,S}}$ and actuation $d^k_{\mathrm{P,A}}$, and at the controller computing the control logic $d^k_{\mathrm{P,C}}$. Computing systems delays arise while *processing* the messages containing sensor data on the robot ($d^k_{\mathrm{P,STX}}$) and on the controller ($d^k_{\mathrm{P,SRX}}$). Actuation messages are processed on transmission (ATX) and reception (ARX), causing the corresponding delays $d^k_{\mathrm{P,ARX}}$ and $d^k_{\mathrm{P,ATX}}$. Finally, network delays (N) can be classified in uplink delay $d^k_{\mathrm{N,S}}$, when sensor values are transmitted, and downlink delays $d^k_{\mathrm{N,A}}$, when actuation commands are transmitted.

In an ideal operation, all the *delays* are bounded and within the sampling period of the control loop. However, in a real implementation, the delays vary according to the chosen software and hardware of the control system, computing system, and communication network. While shorter delays can be compensated with simple techniques, such as busy waiting, higher delays must be carefully taken into account using a proper control strategy. The KPIs capture the most important metrics to analyze and understand the operation of the NCS platform during the benchmarking experiment.

A fundamental aspect that has emerged during the implementation and analysis of the proposed NCS platform is the role of time and *delays* in the system. Delays, i.e., the time needed for information exchange and processing on the robot and the controller, strongly influence the overall performance of the NCS. For this reason, an essential part of the proposed KPIs is relative to time and delays. Delays can arise from control, computation, or communication, with lower delays offering a better service for the NCS. We assess the time and delay-based KPIs by proposing a model of the NCS and by measuring the individual delays presented in Figure 6.5. For each measured delay, additional metrics can be obtained to parametrize a large class of NCS. By calculating the maximum, minimum, mean, and probability density function of the measurement values over a time window, it is possible to characterize the stochastic fluctuations of delay or *jitter* of the connection.

Moreover, *packet loss* additionally affects the operation of NCS. In general, packet loss can occur for several reasons, such as buffer overflows in the OS and in the network elements, or due to transmission errors arising from the physical transmission of the packet. In our NCS, we assume that the network introduces packet loss exclusively and that the event of packet loss additionally

arises whenever a packet experiences a delay higher than a specific *delay upper-bound*. For the delays, additional metrics can be calculated to characterize the stochastic fluctuations of the packet loss.

The *packet rate* and the *bandwidth-delay product* are of minor importance. Our investigated NCS creates less than 1 kpkts/s while transmitting a few sensor or actuator values in each message. Considering the available bandwidth of up to 54 Mbit/s, we did not observe any packet or bandwidth limit. High bandwidth combined with short distances, low packet rates, and equidistant inter-packet gaps, leads to a low bandwidth-delay product. Typically only one packet containing either sensor values or actuator settings was in flight at any given moment during the measurements.

In addition, QoC has a vital role in the system and depends on the physical system and the control logic. QoC KPIs are functions that quantify the evolution of the physical system's state and the controller commands over a time window, i.e., the input and output information of the controller.

6.2 NCSbench Implementation

Section 6.2 is based on a joint publication between Samuele Zoppi, Onur Ayan, Fabio Molinari, Zenit Music, Sebastian Gallenmüller, Georg Carle, and Wolfgang Kellerer [108].

In this section, the implementation details of the proposed open-source NCS benchmarking platform are presented following the architecture of Section 6.1.6. Our NCS platform uses a common IP network and a so-called two-wheeled inverted pendulum robot (TWIPR), a typical platform for NCS experiments [120], [134].

The implementation was developed with a focus on reproducibility. Other research groups should be able to recreate the platform itself as well as the results measured on this platform. Our platform is extensible and adaptable so that it can be deployed for arbitrary research purposes. All the software and hardware components used in the proposed platform are low-cost and highly accessible. Our TWIPR is built using the widely-available and affordable Lego Mindstorms platform, communicates via standard Ethernet and WLAN network interfaces, is open-source, and is written entirely in the Python programming language that is supported by the vast majority of operating and computing systems.

This flexibility allows the proposed platform to be used for the benchmarking of arbitrary NCS. All elements of the NCS architecture of Figure 6.4 can be changed easily: different physical plants can be built using Lego, new control logics can be programmed in Python, arbitrary TCP/IP network interfaces can be connected, and the most popular computing systems and OS can be used.

The description of the implementation is organized as follows. In Sections 6.2.1, 6.2.2, and 6.2.3, we detail the components of the NCS architecture for every CPS domain. Section 6.2.4 describes the measurement of the benchmarking KPIs.

Parameter	Description
Control Application SW	Python 3 open-source controller implementation (cf. Music et al. [135])
Control Physical System	*Gyro Boy* robot of the Lego Mindstorms Education EV3 Core Set
Control HW	DC brushed EV3 Large Servo Motors, EV3 Gyro Sensor
Network Topology$_{\mathrm{A}}$	Two-node network connected via the access point (AP) TP-Link TL841ND (cf. Figure 6.3)
Network Topology$_{\mathrm{B}}$	Two-node network connected via the AP Edimax BR6208AC (cf. Figure 6.3)
Network Stack Controller	Ubuntu 18.04 LTS (Kernel version 4.15)
Network Stack Robot	Debian Jessie (Kernel version 4.4)
Network HW Controller$_{\mathrm{A}}$	Intel 82579LM 1G NIC
Network HW Controller$_{\mathrm{B}}$	ASIX AX88179 1G NIC
Network HW Robot$_{\mathrm{A}}$ *(wired)*	Apple A1277 USB-to-Ethernet dongle
Network HW Robot$_{\mathrm{A}}$ *(wireless)*	Edimax EW-7811Un WLAN USB dongle
Network HW Robot$_{\mathrm{B}}$ *(wired)*	Edimax EU-4306 USB-to-Ethernet dongle
Network HW Robot$_{\mathrm{B}}$ *(wireless)*	Edimax EW-7811Un WLAN USB dongle
Network Physical Env.	Quiet office environment (low interference, no moving objects), indoor, 1 to 2 m distance between robot and controller
Computing Sys. Higher Layers	UDP, application protocol described in Section 6.2.2
Computing Sys. HW Controller$_{\mathrm{A}}$	Intel Core i2520M (2 cores, 2.5 GHz, 8 GB RAM)
Computing Sys. HW Controller$_{\mathrm{B}}$	Intel Core i7-6700 (4 cores, 3.4 GHz, 16 GB RAM)
Computing Sys. HW Robot	32-bit ARM9 SoC (1 core, 300 MHz, 64 MB RAM)

Table 6.1: Scenario description parameters for two NCS platforms A and B—initial implementation and benchmark replication (cf. Zoppi et al. [108])

Furthermore, we summarize in Table 6.1 the scenario description of platform A and a second replicated platform B used in our evaluation. The scenario only presents a minimal description of the basic setup for our TWIPR performing the task of self-balancing. Additional parameters can be added to the scenario description for more complex scenarios. For instance, a TWIPR would require the definition of the path and the surrounding environment. Table 6.2 summarizes the time and control KPI measurements in our implementation.

6.2.1 Control System

The plant is built following the default instructions of the Gyro Boy robot of the Lego Mindstorms Education EV3 Core Set until Step 61 [136]. Figure 6.6 shows the robot's body supported by two wheels, each directly attached to a *DC brushed EV3 Large Servo Motor*. Voltages between -8 and 8 V can be applied to the left and right motor. The voltage applied at a certain point in time t is denoted as $\nu_l(t)$ and $\nu_r(t)$ for the left and right motor, respectively.

An incremental encoder measures the rotation angle of the corresponding wheel. Figure 6.6 presents the rotation angles of both wheels with regard to the z-axis as $\Phi_l(t)$ and $\Phi_r(t)$. Similar to Kim et al. [134], $\Phi(t)$ describes the average rotation angle of the two wheels with regards to the z-axis, i.e., $\Phi(t) = 0.5 \cdot [\Phi_l(t) + \Phi_r(t)]$.

A one-dimensional gyroscope, the EV3 Gyro Sensor, is mounted on the body and measures the pitch rate $\dot{\Theta}(t)$. In Figure 6.6, $\Theta(t)$ is called pitch angle and denotes the angle at time t between the z-axis and the axis passing through the robot's body. Due to the gravitational force, the position $\Theta(t) = 0$ exhibits an unstable equilibrium. Thus, the control goal is a balancing robot, i.e., to hold $\Theta(t) = 0$, while tracking the desired position and orientation in the moving plane

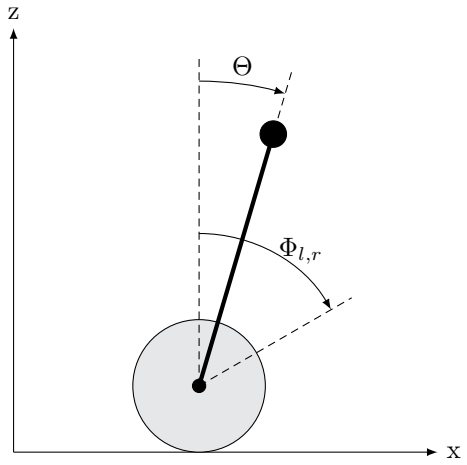


Figure 6.6: Model of the TWIPR, side view (cf. Zoppi et al. [108])

of the robot ($x - y$ plane). This task can be achieved by employing a closed-loop controller, which receives sensor measurements and computes adequate control actions for the two motors. The plant is responsible for the periodic operation of the control loop and regularly triggers sensor readings every T_s. In a real implementation, delays vary according to the chosen software and hardware, which affect the sampling period, and packets can be lost due to high delays or network erasure. Both cases are taken into account by the control logic.

Sensor and actuation data, which are sent over an unreliable wireless network, are subject to high fluctuations. The limited processing capabilities on the robot additionally complicate a reliable and timely data exchange between controller and robot. We employ two complementary strategies to increase control performance. First, we deliberately delay the time for the application of the actuator voltage on the robot. The actuation data is not applied directly after reception but only after a minimum wait time. Experiments showed that we were able to stabilize the robot using a maximum sampling time of 35 ms. The application of actuator data takes up to 6 ms. Therefore, we set the minimum wait time to 29 ms. This time frame can be used to compensate for uncertainties in the time-critical control path, such as fluctuating sensor read times, channel access delays, or L2 retransmits. Second, every actuation message of the controller contains ten additional backup prediction voltages. These backup values can be used if a more recent actuation value does not reach its destination within its expected timeframe of 29 ms. The backup values are calculated on the controller based on its local model. However, without sensor feedback, the modeled backup predictions degrade over time. Experiments have shown that the robot can tolerate up to three consecutive packet losses without falling over.

From the perspective of the control process, the first strategy, introducing an additional delay before actuation, creates a more reliable communication leading to constant sampling and actuation times. The second strategy increases controller robustness by introducing backup values to compensate for lost packets.

6.2.2 Computing Systems

Two different computing systems are deployed in our implementation: one for the controller and one for the robot.

The robot should be mobile and battery-powered, requiring a computing system optimized for compact size and low energy consumption. Any PC available to a researcher should be able to run the controller, i.e., we assume a powerful multi-purpose 64-bit computer and one of the widely spread OS: Windows, macOS, or Linux. Such a controller offers a flexible platform for implementing powerful control algorithms that could not be processed on the resource-constrained robot.

Both computing systems must implement compatible higher-layer communication protocols. For this reason, they use the widely spread TCP/IP network stack of the respective OS. On the application layer, they run a self-developed application protocol. The application protocol consists of two messages: the sensor value message, created by the robot and sent to the controller, and the actuation command message, created by the controller replying to the sensor value message, containing the voltages to be applied to both motors. In addition, sequence numbers and timestamps are transmitted for packet loss or reordering detection, and delay measurements.

6.2.3 Communication Network

Our network is designed to be easily reproducible and flexible concerning possible communication technologies. It is structured according to the OSI communication model and logically separated from the computing system at the network layer, i.e., everything below is part of the communication network.

The *network topology* defines the connectivity of different network nodes at the network and link layers. In our case, a simple two-hop topology is implemented and shown in Figure 6.3. The first hop connects the controller to a WLAN AP via Ethernet. The second hop connects the AP to the robot in two different configurations: wired or wireless network interfaces. In our architecture, the *network interfaces* define the link-layer medium access scheme. The robot has no native network interface, wired and wireless connections are realized via the USB 2.0 interface. This allows switching between the network technology easily. For any given experiment in our analysis, only one of the two connections is used exclusively.

Finally, the *physical environment* describes the physical characteristics of the communication and is particularly important for wireless networks. In our platform, the wireless communication between the robot and the AP takes place in a quiet indoor office environment, at an approximate distance of 1 to 2 m, and it is subject to low external interference.

6.2.4 KPI Measurement

Table 6.2 lists the network-related and the control-related KPIs.

KPI	Description
$d_{\mathrm{P,S}}$	Sensor readings on the robot
$d_{\mathrm{P,C}}$	Calculation of controller's actuation commands
$d_{\mathrm{P,A}}$	Execution of actuation commands on the robot
d_{N}	Average one-way network delay incl. stack processing on robot and controller
$\hat{\Delta}_T - d_{\mathrm{P,A}}$	Robot round-trip delay
$\hat{\Delta}_T$	Measured variable sampling period
Σ_Θ	Total abs. deviation of the pitch angle
Σ_Φ	Total abs. deviation of the wheels' rotation angle
Σ_ν	Total abs. deviation of the average motors' effort
l	Number of controller messages lost or arriving too late at the robot

Table 6.2: Summary of time and control KPIs (cf. Zoppi et al. [108])

Network-Related KPIs

We assess the time KPIs by measuring the individual delays presented in Figure 6.5. To evaluate the influence of the network stack of the controller, we record a packet trace on the ingress/egress network interface via tcpdump.

Recording network delays and performing clock synchronization required a constant packet exchange and increased processing, thus overloading the CPU of the robot and impacting the control performance. Due to this limitation, we did not record the specific delays $d_{\mathrm{P,STX}}$, $d_{\mathrm{N,S}}$, $d_{\mathrm{P,SRX}}$, and $d_{\mathrm{N,A}}$ attributed to the network communication on the robot. Instead, we calculate the average one-way network delay d_{N} assuming symmetrical network delays, and including the stack delays of controller and plant using Equation 6.1

$$d_{\mathrm{N}} = 0.5 \cdot \left(t^k_{\mathrm{A,SRX}} - t^k_{\mathrm{S,STX}}\right). \tag{6.1}$$

As KPI, we report each delay listed in Table 6.2 as median value. We measure the jitter as a property of the delay fluctuation. Low jitter allows a constant stream of information, supporting smooth control performance. To determine jitter, we provide quartiles and 99.9th percentiles in addition to the median delay.

Control-Related KPIs

Section 6.1.4 introduced generic and specific KPIs for the control domain. To keep the benchmarking simple, we do not introduce additional disturbance into our experiment required for the generic KPIs. The experiments presented below observe the robot while it tries to balance on a flat surface. We see this as our baseline scenario, which should be easily replicable by others. In the following, we focus our evaluation on the specific KPIs for our TWIPR. These specific KPIs can be measured more easily, additionally fostering replicability.

We select the *Integrated Absolute Errors* (IAE) of the states Θ and Φ, i.e., Σ_θ and Σ_Φ. Additionally, we calculate the total control effort over time, i.e., Σ_ν.

$$\Sigma_\Theta = \|\Theta(kT_s)\| \tag{6.2}$$

$$\Sigma_{\Phi} = \|\Phi(kT_s)\| \tag{6.3}$$

$$\Sigma_{\nu} = 0.5 \cdot (\|\nu_l(kT_s)\| + \|\nu_r(kT_s)\|) \tag{6.4}$$

Σ_{Θ} and Σ_{Φ} represent the cumulative absolute deviation of Θ and Φ from their corresponding reference values during the experiment. Smaller values of Σ_{Θ} and Σ_{Φ} correspond to a higher QoC. Σ_{ν} represents the total control effort spent to balance the robot. A smaller Σ_{ν} indicates better stability and hence a higher control performance. The control KPIs are summarized in Table 6.2

Concerning our specific implementation of the control logic, we included an additional metric l showing the performance of the control system. The value l is the number of lost or late packets sent from the controller to the robot. This number is equal to the number of predictions used by the robot to compensate for packet loss. As detailed by Music et al. [135], a prediction is applied whenever a packet is not received within the delay upper-bound.

6.2.5 Platform Evaluation

In this section, we provide a comprehensive evaluation of the NCS platform and the benchmarking methodology. We achieve this by presenting the NCS benchmarking KPIs in detail for different scenarios. The evaluation captures the essence of the proposed benchmarking methodology. Experiments were performed to replicate the platform. We test replicability with different computers and networks across two different research groups.

Every experiment of our evaluation is conducted as follows. Before the experiment starts, the robot lies on the ground, continuously sending sensor values to the controller. However, the controller does not send actuation commands until the robot is manually lifted to the vertical position. For this reason, the beginning of the experiment is when the robot manually reaches the vertical position for the first time and corresponds to 0 s in our evaluation.

Afterward, the continuous exchange of information between the robot and the controller takes place and enables the control loop to balance the TWIPR. The control logic determines the duration of the experiment. Our experiments have a duration, T_e, set to 1400 sampling periods, i.e., 49 s with a sampling time T_s of 35 ms. We set the delay upper-bound to 29 ms. This value is smaller than the sampling period and considers the additional time needed to apply the voltage values to the actuators.

Whenever the experiment ends, the controller stops sending actuation messages to the robot, opening the control loop. This way the same number of samples is collected for every experiment, and the KPIs can be correctly calculated and compared.

6.2.6 KPI Evaluation

KPIs belonging to the control, computation, and communication domains need to be evaluated to understand the dynamics of an NCS. The parameters of platform A describe the scenario selected for the detailed evaluation of the KPIs in Table 6.1 communicating over WLAN.

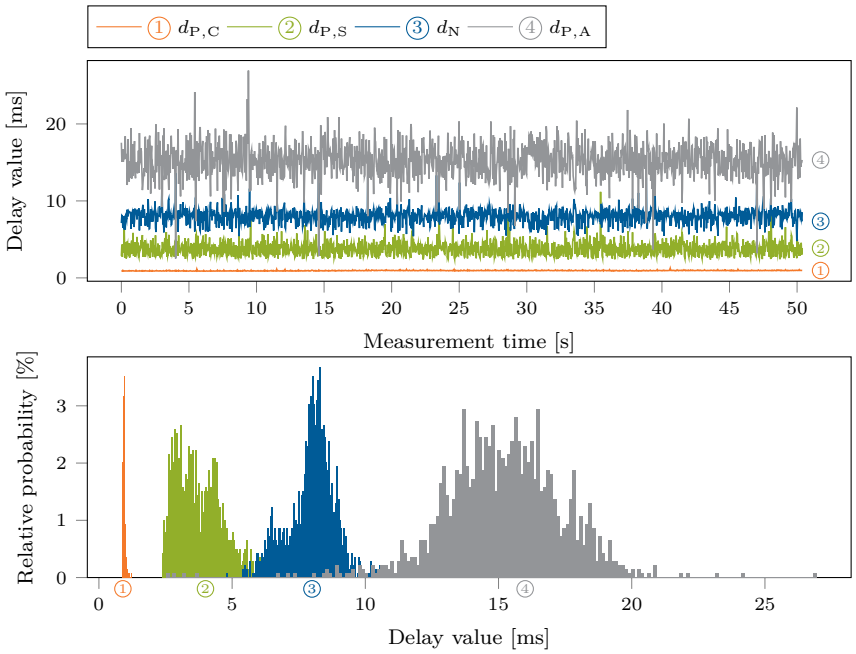


Figure 6.7: Time evolution and empirical distribution of the delays of the controller ①, sensor ②, network ③, and actuator ④ (cf. Zoppi et al. [108])

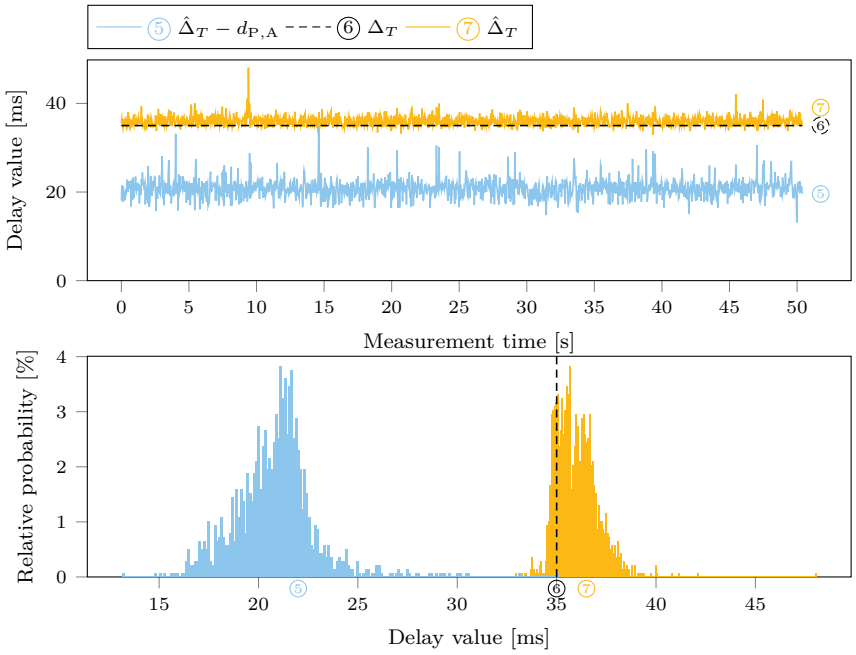


Figure 6.8: Time evolution and empirical distribution of the round-trip delays ⑤, the ideal sampling period ⑥, and of the measured sampling period ⑦ (cf. Zoppi et al. [108])

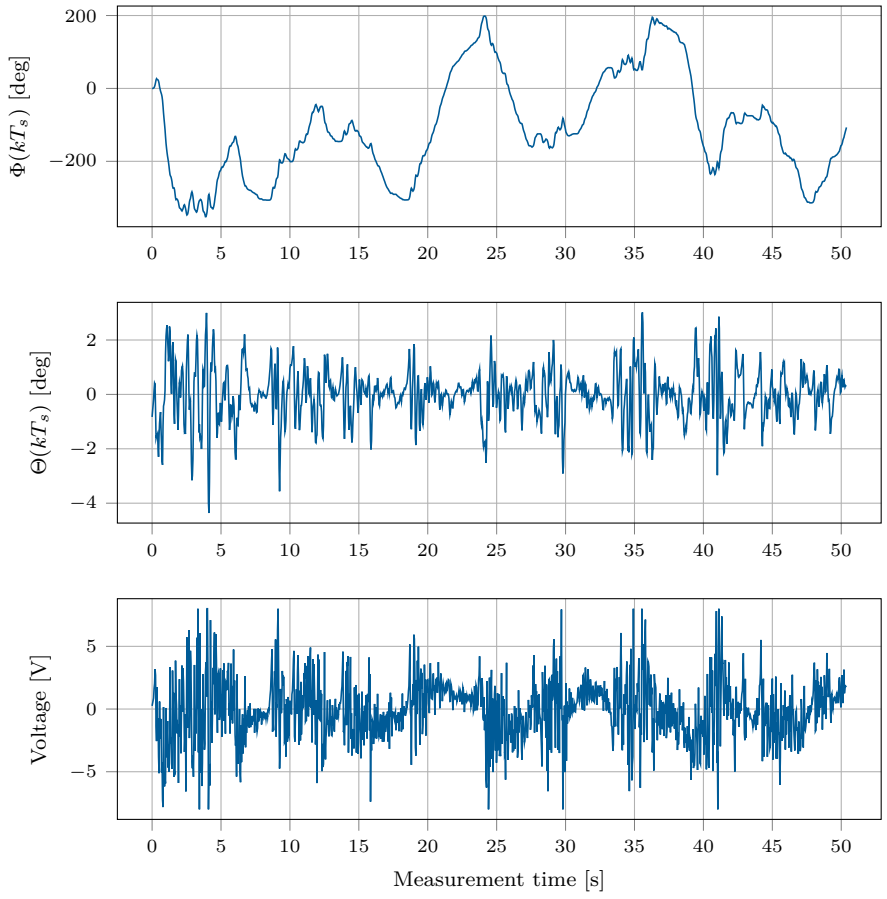


Figure 6.9: Time evolution of the filtered pitch angle Θ, the filtered average rotation angle Φ, and the average applied voltage at the motors ν (cf. Zoppi et al. [108])

Figure 6.7 shows the time evolution and the histogram of the delays of the controller, sensor, actuator, and network defined in Figure 6.5. The sensor reading delay $d_{\mathrm{P,S}}$ ② demonstrates a stable behavior with occasional outliers reaching up to 5.5 ms. Similarly, the controller delay $d_{\mathrm{P,C}}$ ① is stable, showing no outliers. Overhead caused by the controller network stack is constant and marginal (approximately 37 μs) over the entire experiment. The actuator delay $d_{\mathrm{P,A}}$ ④ shows an unstable behavior over the entire measurement period. Its jitter, also expressed by the width of its distribution in the histogram, is attributed to the control algorithm, which implements busy waiting. Therefore, $d_{\mathrm{P,A}}$ ④ includes a waiting period that directly depends on the previous steps and their individual delays. To instruct the motors every 35 ms, actuation commands are only applied after a delay upper-bound of 29 ms from the beginning of the sampling period. In our platform, approximately 6 ms are required to actuate the motors.

When analyzing the jitter given in the histograms, $d_{\mathrm{P,C}}$ ① shows the most stable behavior (0.9 to 1.2 ms), indicating that the controller always has enough computing power to handle the control process on time. The sensor reading delay $d_{\mathrm{P,S}}$ ② shows a minimal time of 2.4 ms for sensor readings with a tail of up to 11.1 ms. The network delay $d_{\mathrm{N,S}}$ ③ roughly resembles a normal distribution ranging from 4.7 to 15 ms, and it is caused by the CSMA/CA mechanism of WLAN in our physical environment.

Figure 6.8 shows the time evolution and histogram of the cumulative delays. The timestamp $t_{\mathrm{A,SRX}}$ is collected by the robot application after receiving the actuation message, resulting in the delay $\hat{\Delta}_T - d_{\mathrm{P,A}}$ ⑤. Where $\hat{\Delta}_T$ is the measured sampling period of the NCS during the experiment. The histogram of ⑤ shows a wide distribution, ranging from 13.2 to 35.1 ms, containing the jitter of all the previous steps. However, if $d_{\mathrm{P,A}}$ is included in the plot ($\hat{\Delta}_T$ ⑦), the jitter decreases, as the actuator algorithm applies the actuation commands only 29 ms after the beginning of the sampling period. This effect results in a rather constant measured sampling period $\hat{\Delta}_T$ ⑦, and allows the compensation of the previous delays, leading to a rather low jitter. Thus, the distribution of $\hat{\Delta}_T$ ⑦ is more compact and allows a constant delivery time for the actuation commands close to the ideal sampling period Δ_T ⑥. Its jitter is caused by the precision of the busy-wait technique and the time required to actuate the motors.

The impact of the control logic is reflected in Figure 6.9, showing the evolution of the control KPIs. The pitch angle $\Theta(kT_s)$ of the robot is highly varying, with occasional larger spikes every few seconds. Despite this, we can observe that its dynamic remains bounded during the execution and that its average value is equal to -0.0014 deg. The evolution of the motors' applied voltage strongly depends on the pitch angle. Higher voltages are correlated with higher values of pitch angle. This effect is also shown in the position of the robot $\Phi(kT_s)$, which presents faster and slower oscillations. Faster oscillations, visible between 3 and 5 s, are caused by strong and opposite actuations commands needed to compensate for high values of pitch angles and balance the robot. Slower oscillations arise whenever the control logic tries to bring the robot to its initial position. This task has a lower priority than balancing the robot, and it is performed on a larger time scale.

Delay [ms]	n-*th* percentiles 50 ±95 % C.I.	25	75	99.9
A-wired				
$d_{\mathrm{P,C}}$	0.94 ± 0.002	0.91	0.97	1.07
$d_{\mathrm{P,S}}$	3.55 ± 0.038	3.04	4.24	5.41
d_{N}	4.38 ± 0.041	4.08	5.03	6.66
$d_{\mathrm{P,A}}$	22.20 ± 0.087	20.86	23.16	24.98
$\hat{\Delta}_T$	35.77 ± 0.042	35.21	36.41	37.73
A-wireless				
$d_{\mathrm{P,C}}$	0.95 ± 0.002	0.92	0.96	1.05
$d_{\mathrm{P,S}}$	3.64 ± 0.049	3.03	4.36	6.20
d_{N}	8.09 ± 0.053	7.54	8.54	10.88
$d_{\mathrm{P,A}}$	15.19 ± 0.118	13.79	16.55	19.94
$\hat{\Delta}_T$	35.89 ± 0.057	35.22	36.62	38.97

(a) Platform A

Delay [ms]	n-*th* percentiles 50 ±95 % C.I.	25	75	99.9
B-wired				
$d_{\mathrm{P,C}}$	0.39 ± 0.001	0.38	0.39	0.45
$d_{\mathrm{P,S}}$	3.89 ± 0.034	3.55	4.49	5.73
d_{N}	4.61 ± 0.026	4.40	4.82	6.57
$d_{\mathrm{P,A}}$	22.38 ± 0.065	21.58	23.10	24.69
$\hat{\Delta}_T$	36.02 ± 0.036	35.55	36.54	37.61
B-wireless				
$d_{\mathrm{P,C}}$	0.37 ± 0.001	0.37	0.38	0.43
$d_{\mathrm{P,S}}$	3.84 ± 0.040	3.49	4.45	6.39
d_{N}	5.25 ± 0.055	4.85	6.29	8.74
$d_{\mathrm{P,A}}$	21.27 ± 0.126	19.49	22.38	24.84
$\hat{\Delta}_T$	36.32 ± 0.049	35.70	36.95	38.76

(b) Platform B

Table 6.3: Time KPI percentiles of the four evaluation scenarios (cf. Zoppi et al. [108])

	Σ_Θ	Σ_Φ	Σ_ν	l
A-wired	762.91	152090	2066.9	0
A-wireless	938.30	217080	2637.4	10
B-wired	601.51	179590	2804.3	0
B-wireless	785.72	129440	2726.1	1

Table 6.4: Control KPIs of the four evaluation scenarios (cf. Zoppi et al. [108])

6.2.7 Benchmarking

We prove the validity of the proposed benchmarking methodology and test the replicability of our platform by conducting experiments in different benchmarking scenarios.

For this, we have built a second Lego Mindstorms robot and tested it in different physical environments. The scenario description of platform B in Table 6.1 contains all the used components. It consists of a different computing system for the controller, and different network hardware interfaces for both controller and robot. The two platforms and the different network configurations result in a total of four scenarios for our benchmarking evaluation. We call A-wired the scenario where platform A operates with Ethernet, and A-wireless its operation with WLAN. Two additional scenarios arise from platform B, B-wired and B-wireless, representing the replicated platform communicating over Ethernet and WLAN.

Tables 6.3 and 6.4 summarize the benchmark KPIs resulting from the evaluation of the four scenarios. The time KPIs in Table 6.3 are presented as median with 95 % confidence intervals, 1st and 3rd quartiles, and 99.9th percentiles.

Table 6.3 shows different performances of the deployed computing systems and communication networks. The median values of $d_{\mathrm{P,C}}$ are lower for platform B than platform A, despite similar jitter and worst-case values. A minor difference is noticeable in the sensor processing delays $d_{\mathrm{P,S}}$; platform A has lower median delays but higher jitter. Additionally, we observe differences in the median network delays. The median of d_{N} is always lower in Ethernet than WLAN.

In addition, WLAN network delays have a higher variance and worst-case delays up to 10 ms. The scenario A-wireless shows the worst network performance, with the highest median value and 99.9th percentile. The actuator processing delays $d_{\mathrm{P,A}}$ directly depend on the busy waiting procedure. Its quartiles reflect the network delays of the wireless setups, fluctuations and the worst case values increase. Finally, the measured sampling period $\hat{\Delta}_T$ is comparable in all four scenarios and mainly depends on the busy waiting performed by the actuator. However, it presents a higher median in platform B, and a larger jitter when operating with WLAN.

Table 6.4 shows comparable values of QoC, for the two NCS in the four evaluated scenarios. In general, Σ_Θ and Σ_Φ are lower in wired than wireless scenarios thanks to lower median delays and jitter. However, platform A shows a high value of Σ_Φ caused by the high oscillations introduced by the delays of its WLAN network interface. The total controller effort Σ_ν is similar across the scenarios, showing a lower value only in scenario A-wired. As expected, actuation predictions on the robot, triggered by packets arriving later than 29 ms, were not observed in wired scenarios. However, in the scenario A-wireless, 10 prediction events were observed, and, in the more stable scenario B-wireless, only 1 event was observed, demonstrating its superior QoC.

The proposed KPIs can highlight the differences in performance of the two computing systems and network interfaces. The platform was replicated across two research groups and used for benchmarking. Results between the two replicated platforms do not match exactly due to differences in the hardware and the physical environment. However, we can identify the same trends despite the differences in values, e.g., we measured the highest variance in the wireless network for both platforms. These results prove the value of the proposed NCS platform and the validity of the benchmarking methodology.

In Section 6.3.1, we want to investigate if and how we can improve our measurement capabilities to increase repeatability and replicability for WLAN experiments.

6.3 Repeatable Wireless Measurements

Section 6.3 is based on joint work between Sebastian Gallenmüller, René Glebke, Stephan Günther, Eric Hauser, Maurice Leclaire, Stefan Reif, Jan Rüth, Andreas Schmidt, Georg Carle, Thorsten Herfet, Wolfgang Schröder-Preikschat, and Klaus Wehrle [137].

End-to-end latency is a relevant KPI for network applications. Content delivery networks (CDN) are an established technology to bring the content closer to the users, thereby shortening the distance and ultimately the delay between the user and content. Edge computing generalizes this concept by providing not only caching of web content like CDNs, but also by providing distributed computing resources. By bringing the compute resources closer to the *edge* of the network, i.e., closer to its users, delay decreases [138]. This reduced distance allows realizing applications depending on low latencies, such as NCS.

Edge computing is a control-agnostic approach where an application moves to a different location in the network to meet the operating conditions for an NCS. The counterpart to this solution would be to adapt the control algorithm to the current network conditions for running an NCS.

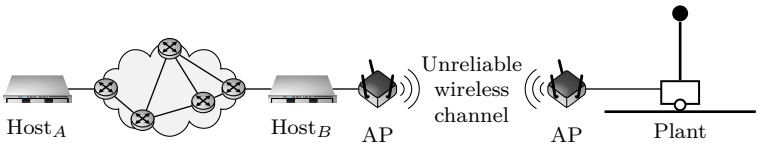


Figure 6.10: NCS topology (cf. Gallenmüller et al. [137])

Gain scheduling is such an approach, where the used control algorithm is chosen from a family of controllers depending on the current conditions of the network—a common approach for control systems [139].

Common to both approaches—edge computing and gain scheduling—is the need for close observation of the available network operating conditions. Therefore, we investigate a protocol specifically designed to measure and report network conditions to the application. We start by illustrating a typical NCS (Section 6.3.1) and present related work on control over wireless networks (Section 6.3.2). We then design and implement a prototypical framework that enables the live monitoring of delays and channel conditions of a wireless network based on the combination of an instrumentable realtime transport protocol (Section 6.3.3) and an automated shielded testbed for wireless communication (Section 6.3.4). Afterward, we provide an empirical evaluation of our approach to the collection of channel state information (Section 6.3.5).

6.3.1 System Model

Figure 6.10 shows a physical system or plant attached to a network via a radio link. Two servers, Host$_A$ and Host$_B$, are connected to the network differing in their distance (number of hops) to the plant. Adding a control application, running on either of the hosts, turns this into an NCS. The plant itself and the inherent properties of the actual control process determine the network connection requirements, such as maximum delay, number of exchanged messages, and maximum packet loss.

We propose installing the control application as close as possible to the controlled plant to reduce potential network delay or avoid network bottlenecks. In the case of the system in Figure 6.10, we would prefer Host$_B$ over Host$_A$ for running control applications due to the lower distance from the plant. Despite shortening the link between the host and the plant, network behavior may change over time, especially if wireless links are involved. Gain scheduling allows us to react to these changes by selecting a control application that fits best to the current operating conditions of the underlying network. In this work, we do not provide a complete network control system that dynamically adapts to rapidly changing network conditions and solves all issues of NCS in general. Instead, we focus on two critical aspects of such a system: First, we investigate a protocol equipped with in-band live-monitoring features that can be used to collect the information necessary for gain scheduling (Section 6.3.3). Second, we evaluate how this protocol behaves on a wireless link through a series of reproducible network experiments (Section 6.3.5). Due to the unreliable nature of wireless links, this part of the network connection is the most challenging component of the control system.

6.3.2 Related Work

This section analyzes the challenges when combining the network and control domain in NCS. Costa et al. [140] investigate different QoS schemes for WLAN according to IEEE 802.11. Simulations show that none of the available techniques supports realtime traffic, even for scenarios with low network load. Therefore, various strategies were proposed to utilize wireless networks for control systems. Nakashima et al. [141] propose the co-design of network and control when realizing an NCS. They propose a time-division multiple access (TDMA) strategy to create a deterministic network behavior and consider propagation times of the network for their controller design. Nikolakopoulos et al. [142] use WLAN connections in conjunction with gain scheduling to create a robust NCS. Xia et al. [143] utilize a co-design of network and control. Their design adapts the sampling period of the control process to enable a suitable QoS under changing network conditions.

Common to the mentioned works and our contribution is the idea of co-design between network and control. However, previous work primarily relies on simulation for evaluation, assuming specific hardware behavior. Our work performs experiments on real hardware, which allows us to analyze a realistic behavior between hardware, its driver, and the OS in a typical control scenario.

The indeterministic behavior of the wireless links makes repeating experiment results challenging. To gain repeatable results, we perform our experiment in a shielded environment and apply the pos experiment workflow (cf. Section 2.4) to this wireless testbed. We see our work as a first step towards a fully reproducible real-world analysis of NCS.

6.3.3 Design and Implementation

We create a runtime support system observing link behavior to enable gain scheduling in wireless networks of NCS. As a basis, we use the openly available predictably reliable realtime transport (PRRT) protocol [144], [145], which provides partial reliability and in-order delivery, and at the same time, allows making statements about the timing characteristics. The timing behavior is influenced by the requirements of an application, such as the maximum tolerable latency. Thereby, one controller instance, designed for a specific latency, can communicate this requirement to the runtime system and the protocol.

Traditionally, only the control application—but not the network stack—is aware of latency requirements that are a constraint of the physical process it is designed to control. IP-based control applications can choose between two services: First, they can use a fully reliable transport protocol, such as TCP or QUIC, which retransmits messages even if the latency demands cannot be met any longer. Second, it can use an unreliable transport protocol, namely UDP, which does not retransmit even if latency demands would allow it. PRRT allows an application to use a hybrid service combining features of both protocol families, providing *partial reliability* with *predictable timing*. Using the PRRT stack, the application passes its latency requirements to the network stack, which can handle retransmissions while respecting latency requirements. If the latency requirements of a message cannot be met any longer, PRRT discards it—thereby avoiding a waste of time and energy.

Naturally, there are operating conditions that do not allow for the fulfillment of these constraints, e.g., meeting a 1 ms end-to-end deadline on a wireless link with 5 ms propagation time. In these cases, PRRT makes this issue transparent to the application, letting the application pick remediation, e.g., triggering emergency routines or adapting its general control strategy. In the case of gain scheduling, this *notification* is the latest point in time to switch to a different controller instance that can handle the current conditions. A more efficient way to trigger the controller switch is to continuously probe the runtime system for a change in observed latencies or register an event handler.

As long as the operating conditions allow for PRRT to fulfill its requirements, PRRT uses two techniques to do this reliably and predictably: (a) error control and (b) a combination of congestion and rate control.

Error control is implemented as a block-based hybrid ARQ scheme, so PRRT aggregates multiple packets to a block. The packets themselves are sent out as fast as they arrive and proactive redundancy is sent as soon as a block is filled. Afterward, reactive transmissions of redundancy are triggered if no acknowledgments arrive within a round-trip time plus processing margin. The arrival of a sufficient amount of data or redundancy packets for a block allows reconstructing all packets of the block, e.g., previous sensor readings or actuator inputs. While the relevance of any sensor reading older than the latest is zero for Markovian controllers, our solution targets controllers where either (a) there is no Markovian model and the history is important (e.g., to detect temperature trends) or (b) the controller is fitting such a model during operation. Using error control, the protocol can optimize resilience under the given latency constraints.

Congestion and rate control minimize queuing that would lead to excessive delays by controlling both the amount of data in flight and the rate of packets. This combined approach aims to avoid both self-induced and contention-based queueing delays, a well-known problem of loss-based TCP congestion control [146].

The controller-supplied latency constraint is further used as a deadline for messages, i.e., messages that have already exceeded the deadline or are going to exceed it with certainty are not processed further. Thereby, perturbations of the end-to-end latency that lead to a single packet not arriving in time do not impede subsequent packets that can still make the deadline. Additionally, the `recv()` calls have a *receive_window* parameter to filter packets that are ready to be delivered, namely those that expire in between *now* and $now + receive\_window$.

This timing awareness within PRRT enables our runtime system to select controller instances dynamically depending on the current operating conditions. The cooperation between the transport layer and the control application is hence symbiotic: the transport protocol provides timing measurements for controller selection while the control application dynamically reconfigures latency and, indirectly, error control parameters.

PRRT measures the network round-trip time using an algorithm similar to NTP by including timestamps in its metadata and feedback packets and compensating for processing time. Simultaneously, PRRT tracks the current data rate by estimating the delivery rate on the sender side, leveraging a mechanism presented in an IETF draft [147] from 2017. The implemented

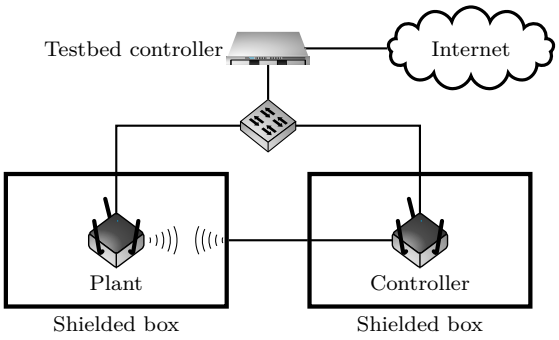


Figure 6.11: Simplified testbed setup (cf. Gallenmüller et al. [137])

congestion control follows the design of BBR [148], including adaptations of recent fixes in the Linux kernel code for TCP-BBR. This congestion control, together with rate control through packet pacing, aims to avoid queueing at all stages of the communication, minimizing latency and jitter.

In summary, our runtime support system exploits PRRT, a partially reliable and latency-aware transport protocol, for gain scheduling. It enables control applications to adapt dynamically to the currently faced communication latencies. Simultaneously, the controller selection allows the transport protocol to minimize the error rate as well as jitter.

6.3.4 Testbed and Measurement Setup

To perform comparable benchmarks of our runtime system in different settings, a reproducible test environment is essential. Since wireless networks are affected by many different factors such as noise, networks on the same or neighboring channels, fading channel conditions, and radar detection, it is challenging to guarantee comparable conditions across different measurements. Note that we use this testbed setup to ensure the repeatability of our evaluation runs, but it is not required to operate our system.

To allow comparisons between benchmarks of PRRT at different settings, we use the setup depicted in Figure 6.11, consisting of two wireless test nodes (*plant* and *controller*) placed in shielded boxes. For these experiments, we use a small computer as plant, not an actual robot due to space constraints of our shielded boxes. The antenna port of the controller is connected to a shielded coaxial cable that is connected to an antenna placed in the plant's shielding box. An air gap between that antenna and the antenna of the plant within the shielded box ensures constant channel conditions resembling an undisturbed wireless network allowing for repeatable wireless conditions across all measurements. We use IEEE 802.11g (54 Mbit/s) in ad-hoc mode and generate PRRT packets with a constant sampling interval. The test nodes use Debian Linux (kernel version 4.8). They are equipped with AMD GX-412TC CPUs (4× 1 GHz "Jaguar" cores), Qualcomm Atheros AR958x IEEE 802.11abgn wireless network adapters, and Intel I210 NICs.

Both nodes are connected via Ethernet to a *testbed controller*, which is connected to the Internet for remote testbed operation. The testbed uses the pos framework [3] to execute the network experiment. It orchestrates a set of measurements by performing the following steps:

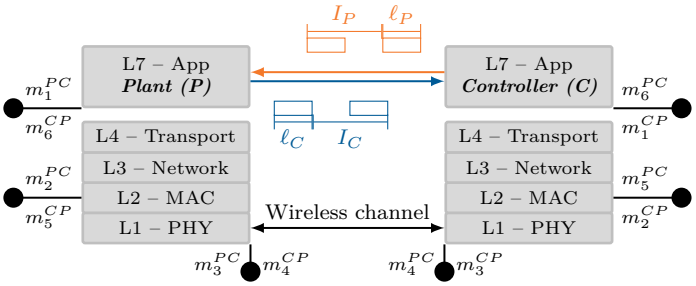


Figure 6.12: Stack with timestamping vantage points (cf. Gallenmüller et al. [137])

1. After each measurement run, the test nodes are completely reset by power cycling and booting a live system via PXE from the testbed controller, eliminating any residual effects of the previous measurement run, such as the firmware of wireless devices being initialized with unwanted settings.
2. When the nodes are booted, clocks are synchronized once via PTP utilizing the hardware support of the I210 NICs. This synchronization is crucial to obtain comparable timestamps on both nodes. Starting with a deviation below 1 μs after synchronization, the clocks' deviation does not exceed 10 μs after a single test run of 2 min.
3. When all preparations are finished, the test nodes are ready to execute the actual measurement software.

The software works with two independent threads: the first thread only transmits and receives packets, while the second thread captures packets using *libpcap*. This architecture ensures that packets are processed as soon as they are received to keep the timestamps as accurate as possible.

In order to evaluate PRRT, we integrated it into our measurement software, which allows recording timestamps at various locations throughout the protocol stack, as shown in Figure 6.12. Considering the direction from plant to controller (denoted as PC), we obtain the timestamps

- m_1^{PC} when the app transmits a message,
- m_2^{PC} when a frame becomes visible by libpcap at the transmitting node, i.e., before it is transmitted,
- m_3^{PC} when the *echo frame*[1]—including the sender's radiotap header—becomes visible at the transmitting node,
- m_4^{PC} when a frame is received,
- m_5^{PC} when the received frame becomes visible through libpcap at the receiving node, and
- m_6^{PC} when the measurement software receives a message.

[1] By *echo frame*, we refer to the frame including the radiotap header provided by a wireless card's driver when a frame has been transmitted successfully.

Parameter	Minimum	Maximum	Steps
Packet-to-packet time (I)	1 ms	10 ms	5
Payload size (ℓ)	20 B	1400 B	5
PRRT target delay	1 ms	10 ms	5
PRRT receive window	0.1 ms	2 ms	6

Table 6.5: Parameters of the delay measurement set (cf. Gallenmüller et al. [137])

The radiotap header thereby contains various information about how a frame has been transmitted, e.g., the chosen transmit rate. The same holds for the reverse direction (denoted as CP). We record both directions separately to investigate the potentially different behavior of the WLAN connection. Using these timestamps, we can derive delays that are difficult to determine under ordinary circumstances. For instance, $m_6^{PC} - m_1^{PC}$ is the one-way delay from plant to controller. This one-way delay is of particular interest for the evaluation of PRRT as it allows to verify whether or not datagrams are within the defined receive window. Similarly, the delay $m_3^{PC} - m_2^{PC}$ is primarily influenced by the media access time, which can be determined precisely when the serialization time of frames is known.

If PRRT cannot deliver a packet within the desired interval due to delays on the wireless channel or within the OS, the packet is discarded. If a packet is discarded for that reason on the way from the plant to the controller, the timestamp m_6^{PC} is missing as PRRT dropped the respective packet due to a violation of the desired time interval. The same holds for m_6^{CP} when a packet in the reverse direction is dropped. The timestamps between m_1 and m_6 are useful for investigating the channel's characteristics or influences of the OS independently from PRRT. They can help comprehend why PRRT, for instance, could not deliver specific packets in time. The delay $m_2^{PC} - m_1^{PC}$ gives an insight into how long PRRT needs to process packets from the application and hand them over to the network stack. Correspondingly, the delay $m_6^{PC} - m_5^{PC}$ shows how long packets are delayed before being handed back to the application.

After a measurement run has finished, the files from the test nodes containing the timestamps are copied to the testbed controller and the parameters of the test run are logged.

6.3.5 Evaluation

We investigate the behavior of PRRT through a series of measurements using a combination of four configuration parameters: the packet-to-packet time (I) specifies the sampling time of a control process, the payload size (ℓ) sets the data transmitted by the control process, the PRRT target delay defines the time data should arrive at the control process, and the PRRT receive window defines a grace period (cf. Section 6.3.3). Table 6.5 contains the values for the measurement parameters. All possible combinations result in 750 distinct measurement runs. Despite almost identical channel conditions, we measured different behavior for both directions of the communication channel. Therefore, we present our measurements for both directions separately. The following measurement investigates the two main network-related KPIs relevant for control systems, latency and packet loss. We consider packets, which arrive late at their destination as lost.

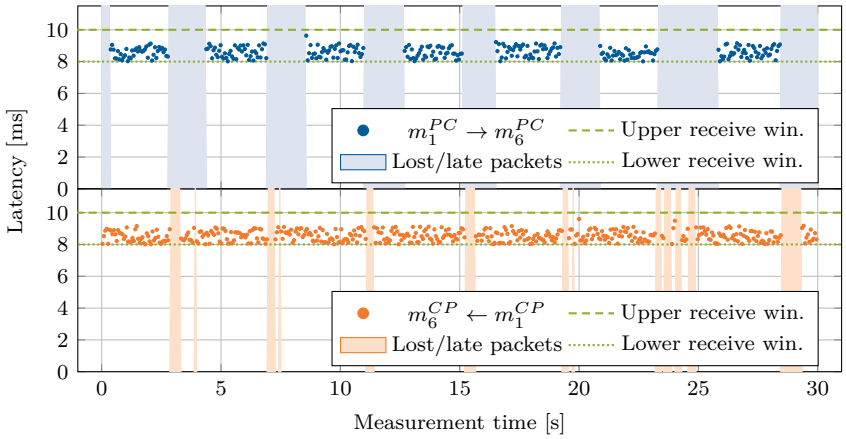
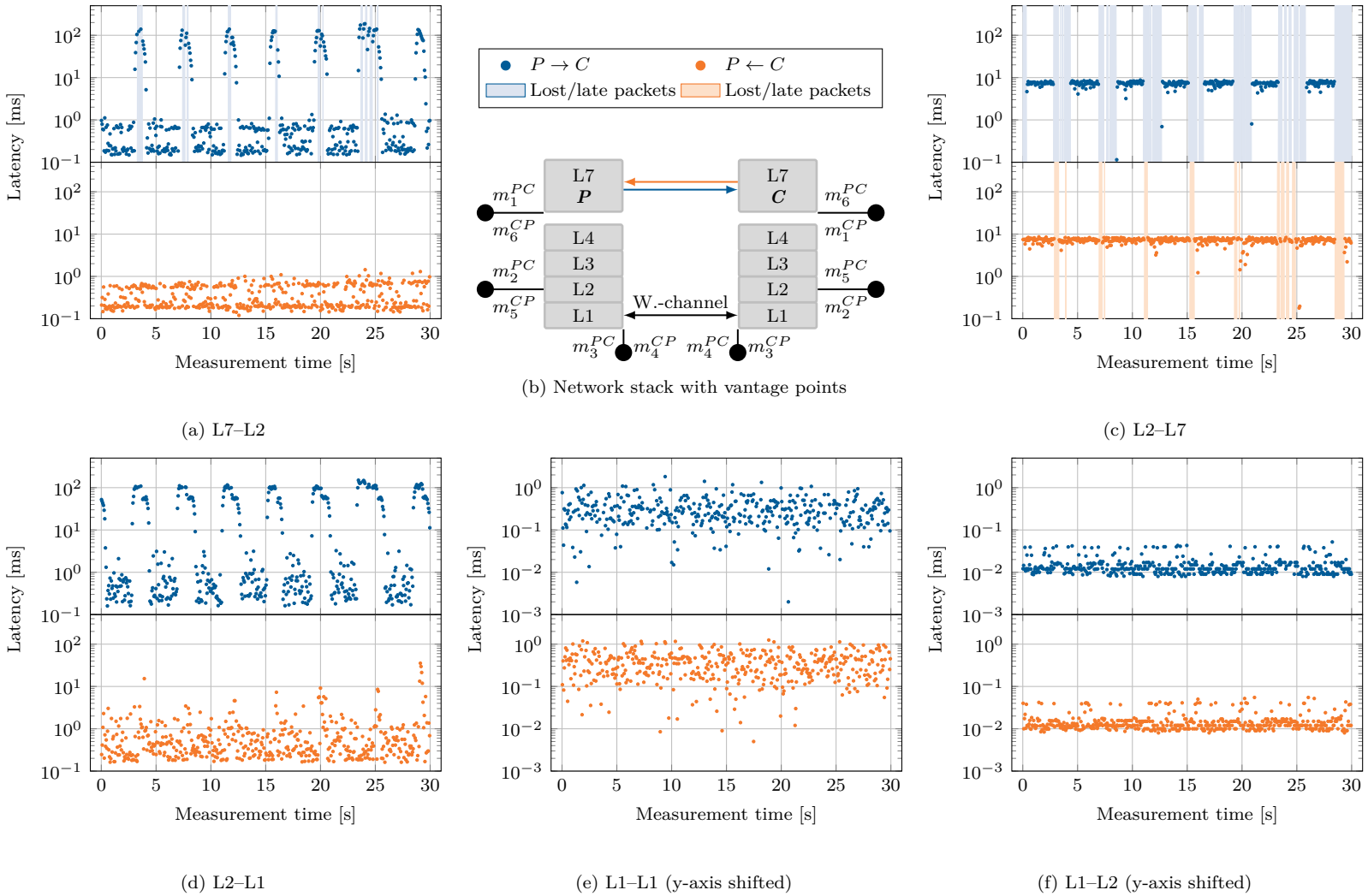


Figure 6.13: 1 ms packet-to-packet time, 20 B payload size, 10 ms target delay, 2 ms receive window (upper and lower limit in green) (cf. Gallenmüller et al. [137])

Packet-to-packet time: Our first measurement measures the impact of the packet-to-packet time. Therefore, we select the measurement with the most demanding setting for the packet-to-packet time, i.e., 1 ms, while we relax on all other parameters. We chose the smallest payload size of 20 B, a wide receiving window of 2 ms, and allow a target delay of 10 ms. Figure 6.13 shows this measurement as a time series over 30 s for PC and CP direction. Both plots show the delay from the respective sending application to its destination ($m_1 \rightarrow m_6$) as scatter plot. With this parameter configuration, it is possible to transmit packets within the specified target delay and receive window visualized by the two green lines. However, for both directions, there are periods of up to 2 s without any packet delivered on time marked by the shaded areas. We observe that losses for both directions typically start simultaneously, which hints at a common root cause for the packet loss. We observe that the CP direction recovers faster than the reverse direction.

Figure 6.14 visualizes the delay measured at different vantage points of the network stack—on the controller and the plant. This allows a detailed investigation if packets are lost or dropped due to specific deadline misses by PRRT. Figure 6.14a shows the delay caused by packet processing after the packet has left the application until the driver accepts the packet. The delay in CP direction stays below 2 ms. The PC direction already shows that several packets are not received at L2 and that there are packets with a delay higher than the configured target delay of 10 ms. We attribute the packet loss to buffer overflows, not to an intentional decision of PRRT, as observed behavior is consistent with typical buffer overflow behavior—buffers begin filling up, increasing the measured delay. If buffers are not drained fast enough, packet loss occurs. Occasionally, high-delay packets are transmitted during phases of loss. If buffer overload decreases, packet delay decreases to its original value. We see that pattern repeating in Figure 6.14a.

Packet processing continues in the driver (L2–L1, cf. Figure 6.14d). There, the target delay is violated for a number of packets in both directions, but no additional packet loss occurs. Periods of high delay (above 100 ms) roughly coincide between driver and higher layers, but these periods start earlier and end later in the driver. This indicates that the driver cannot process the packets fast enough, propagating these problems up to the higher layers.



(a) L7–L2

(b) Network stack with vantage points

(c) L2–L7

(d) L2–L1

(e) L1–L1 (y-axis shifted)

(f) L1–L2 (y-axis shifted)

Figure 6.14: Layered WLAN measurement

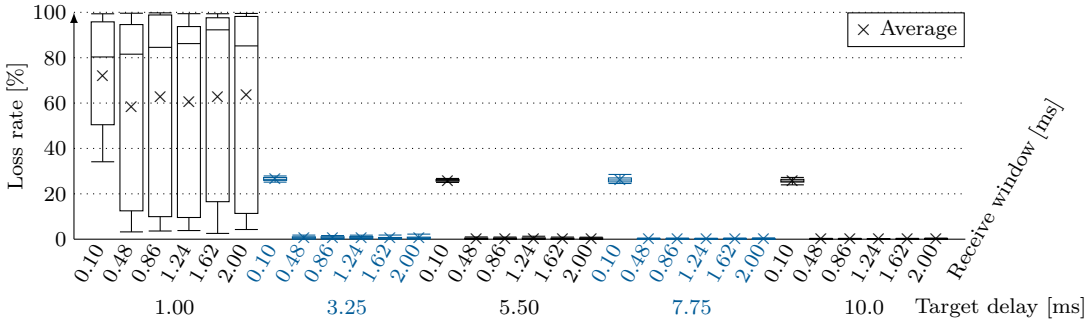


Figure 6.15: Measurements of PRRT with varying target delay and receive windows (cf. Gallenmüller et al. [137])

Figure 6.14e shows the delay used for transmitting the packet on the medium and processing steps happening shortly before or after. The distance of 1.5 m causes a propagation delay in the nanosecond range. Delay is almost consistently below 1 ms, with both directions behaving similarly. We do not see significant effects of jitter caused by buffers as for the higher layers. In the next step, shown in Figure 6.14f, the driver of the respective communication partner receives the packet and transmits it to the network stack. This is the fastest processing step causing a roughly constant delay of 0.1 ms or lower for both directions.

Figure 6.14c shows the processing of the PRRT network stack on the receiver side. This processing step causes a delay of up to approximately 9 ms. Here, PRRT causes the delay by intention to meet the configured target delay requirements. In addition, PRRT drops packets that fail to meet the target delay, leading to many packet drops for both communication directions.

Adding up the losses caused in Figures 6.14a and 6.14c results in the loss pattern observed in Figure 6.13. For our measurement, we observed losses only from L7 to L2. We did not see any packet loss or drops between the communication on L2 or lower on any host. All other packets were discarded intentionally from L2 to L7 on the receiving host because of target delay limitations. The packet rate in this experiment was limited to 1 kpkts/s. Despite this low load, the measurement showed typical overload behavior of packet loss and high delays. We identified the transition from the transmitter to the medium as the main bottleneck of the connection. For Ethernet, we did not see such a bottleneck in any measurement. Considering the medium usage, WLAN differs significantly from Ethernet. WLAN uses a more complex access scheme for its shared medium and it uses only a half-duplex mode. Both differences contribute to the creation of this additional bottleneck for wireless connections. A gain scheduling approach can use that information to avoid this bottleneck, selecting a controller instance that operates at a lower sampling frequency to allow stable control performance over WLAN channels.

We are interested in the behavior of non-overloaded systems. Therefore, we increase the packet-to-packet time to at least 3.25 ms for the following experiments.

PRRT target delay and receiving window: Figure 6.15 shows the loss rate over a series of measurements with varying target delay and receive windows. Loss rates for a target delay of 1 ms have a median above 80 %. Figures 6.14a and 6.14d can explain these high loss rates. The two processing steps alone cause a delay close to or even above the configured target delay of

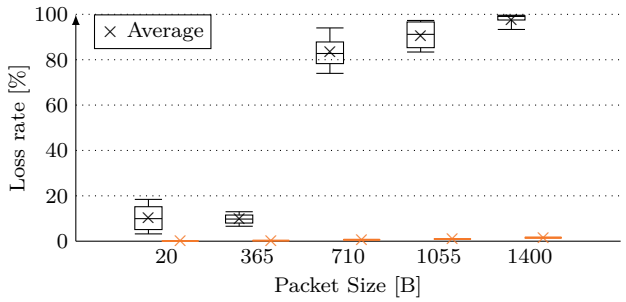


Figure 6.16: Measurement of PRRT for different packet sizes, receive window of 480 μs, target delay of 1000 μs (black) and 3250 μs (orange) (cf. Gallenmüller et al. [137])

1 ms. A narrow receive window also influences the loss rate: for 0.1 ms, the loss rate reaches 25 % independent of the target delay. For wider receive windows and target delays above 3.25 ms, the loss rate stays below 3 %. These results indicate that the achievable target delay has the highest impact on the loss rate. However, even if high target delays are combined with narrow receive windows, packet loss may remain high. If the packet loss rates are unacceptable for a controller instance, gain scheduling can switch to another controller instance, respecting both target delay and receive window sizes.

Payload size: To investigate the influence of the payload size on the loss rate, we pick two examples from Figure 6.15 for a closer investigation. Figure 6.16 shows an example for a receive window size of 0.48 ms and a target delay of 1 ms in black. Only for small payload sizes of 20 and 365 B a loss rate below 20 % can be achieved. For larger payload sizes, the loss rate steeply rises to over 80 %. The second example in Figure 6.16 shows the scenario with a receive window of 0.48 ms and a target delay of 3.25 ms in orange. There, the packet size has only a minor influence on the loss rate, rising to 1.75 % in the worst scenario using a payload size of 1400 B. As the packet size has only a minor influence on the loss rate—compared with the previous parameters—gain scheduling should consider packet size as a minor input factor.

Our measurements identified the sampling time, target delay, receive window, and payload size, all impacting the packet loss over IEEE 802.11g networks. In the following section, we try to model the impact of these factors on network throughput. Therefore, we investigate if our resource model for wired networks is also applicable to WLAN.

6.4 Applying the Resource Model to WLAN

We chose the same platform we used for our repeatable WLAN measurements to investigate the applicability of the resource model to WLAN. This platform offers a low power CPU (AMD GX-412TC CPU, 4× 1 GHz) that could be used for a mobile resource-constrained CPS. We further chose a WLAN operating according to IEEE 802.11g, which offers a data rate of up to 54 Mbit/s. The data rate is plenty for a typical NCS, e.g., the TWIPR of NCSbench, which requires approximately 30 pkts/s containing less than 1 kbit per message. We further use the

ad-hoc mode for WLAN to simplify the network topology by removing the need for a WLAN access point.

Utilizing the same system architecture, WLANs are subject to the same bottlenecks as wired Ethernet (cf. Section 4.1). Our chosen platform uses a single lane of PCIe 2.0 that offers a bandwidth of 4 Gbit/s. The DDR3-1333 memory offers a bandwidth of approximately 85 Gbit/s. Both bandwidths exceed the maximum bandwidth of 54 Mbit/s for IEEE 802.11g networks by far. Therefore, neither system interconnect presents a relevant bottleneck to our chosen system.

The main bottleneck of packet processing on high-performance systems is the CPU. To check if the CPU can act as a possible bottleneck, we performed measurements. We used our typical two-node setup relying on the hardware mentioned above in our shielded environment under optimal conditions. Iperf3 was the packet generator of choice, as the WLAN NIC does not support DPDK or MoonGen. One of the nodes acted as a traffic source, the other node as a traffic sink. We did not observe a CPU load higher than 20 % at any point during our various measurements, neither on the receiving nor the sending node. Therefore, we conclude that the CPU is not a bottleneck for the investigated packet rates in our scenarios.

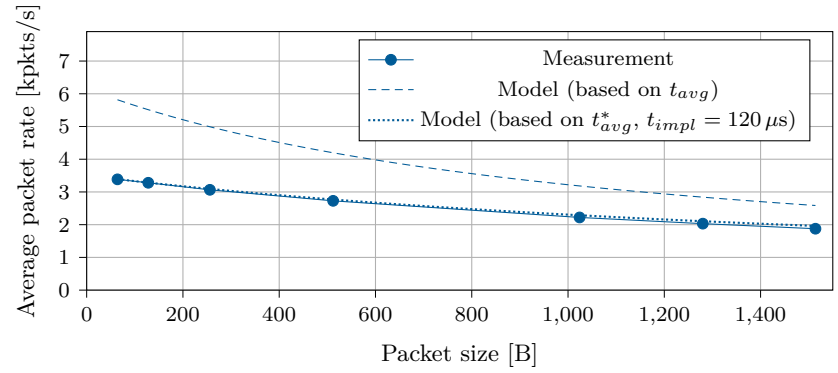
The only remaining bottleneck is the bandwidth of the WLAN technology in use. WLAN bandwidth depends on many factors, such as the used encoding scheme or the length of the transmitted packets. Bordim et al. [149] provide equations for calculating the available bandwidth considering these factors.

$$t_{avg} = 0.15x + 162.43 \tag{6.5}$$

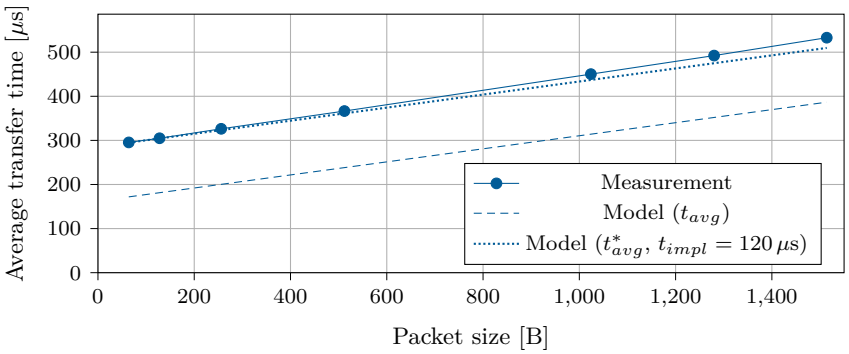
Equation 6.5 models an ad-hoc network with an encoding scheme allowing the highest bandwidth of 54 Mbit/s. This equation calculates the average time to transmit a packet in μs depending on the packet size x in B. Equation 6.5 demonstrates that IEEE 802.11g has average per-packet costs of approximately 162 μs and per-byte costs of approximately 0.15 μs. Figures 6.17a and 6.17b, show the throughput and the transmission time per packet according to this model as dashed lines.

To test the applicability of this model to a real-world scenario, we performed our own measurement. We use the hardware listed above and the iperf3 packet generator, creating UDP traffic to saturate the link. Unidirectional traffic was sent between a traffic source and a traffic sink equipped with identical hardware. Figure 6.17a shows the measured throughput as a solid line. When comparing measured and modeled values in Figure 6.17a, the measured throughput is always lower than the modeled one. However, this performance penalty seems to decrease for larger packets. To compare the measurements directly with Equation 6.5, we calculated the average transmission times for our measured throughput figures. The result is depicted as a solid line in Figure 6.17b.

Bordim et al. [149] also performed measurements and noticed a difference between model and measurement. They attribute this difference to various factors, such as hardware and software implementations. In their setup, they determined a constant additional delay of approximately



(a) Throughput



(b) Per-frame transmission time

Figure 6.17: Impact of frame size on IEEE 802.11g WLAN

60 μs. For our measurement, we calculated a constant delay of approximately 120 μs. The measurement uses the RF-shielded boxes to provide optimal conditions for WLAN. Therefore, we attribute the differences between the two measurements to the differences in the hardware and software. To improve the prediction of Equation 6.5, we include this additional delay as an implementation-specific variable t_{impl} resulting in an improved model noted in Equation 6.6.

$$t^*_{avg} = 0.15x + 162.43 + t_{impl} \tag{6.6}$$

This improved model can be used to determine the per-packet transmission time on a wireless link for a specific implementation of WLAN hardware and software. The model requires the additional delay t_{impl}. This variable can be determined by a throughput measurement at the maximum available bandwidth. As t_{impl} remains constant for a given implementation, Equation 6.6 can calculate an average transmission time for the wireless link. This average transmission time can further be used to calculate the maximum packet rate (T^{WLAN}_{max}) of a given wireless link using Equation 6.7.

$$T^{WLAN}_{max} = \frac{1}{t^*_{avg}} \tag{6.7}$$

The original model and the different measurements uncover substantial differences in transmission times for WLAN networks. Therefore, we recommend measuring the WLAN throughput in a specific scenario to determine the impact of the implementation expressed by t_{impl}. Using this implementation-specific variable, Equation 6.7 can be used to provide a realistic upper bound for the achievable throughput on a wireless link. We added an improved model that respects t_{impl} with a value of 120 μs in Figure 6.17. This new model can provide realistic upper bounds for WLANs under the given hardware and operating conditions.

These results are consistent with our measurements in Section 6.3.5. The sampling time and, therefore, the packet rate and the packet length impact the achievable packet rate, which can be explained by Equations 6.5 and 6.7. The impact of the other two factors, the receive window and the target delay, cannot be explained by these equations. Both are a result of the system architecture of Linux that cannot process the packets with the low delay needed, despite the availability of enough CPU time.

Our model considers two impact factors, the IEEE 802.11g standard and its implementation. The measurements demonstrate that both impact factors are essential to determine the link capacity. Therefore, this model can be used as a foundation for more complex scenarios. Such scenarios may require additional factors, such as changing operating conditions, e.g., the distance or the angle of the WLAN antennas.

6.5 Key Results

This chapter demonstrates the application of a data-driven measurement approach towards the domain of networked control systems (NCS). Initially, we present a novel benchmark suite, called

NCSbench, based on benchmarks from both disciplines joint in NCS, namely the control domain and the network domain. This allows us to create a new benchmark tailored to the specific needs of an NCS. Therefore, we identify the relevant parameters to recreate test conditions when performing the benchmark. The results of our benchmark are reported as KPIs. We define two sets of indicators that are either used to describe the QoC of a CPS or the quality of transport of the underlying network.

NCSbench includes an open-source NCS experimental platform based on the Lego Mindstorms. We evaluate the platform by repeating our own results and replicate results among two research groups. The evaluation results prove the effectiveness of the proposed KPIs and the validity of the benchmarking methodology. The platform is built with cost efficiency, flexibility, and ease-of-use in mind. These design choices lower the barriers for others to recreate our experiments, thereby creating full reproducibility.

Our NCSbench evaluation shows that, despite our efforts for documenting and replicating the measurement setup, the results of wireless experiments remain highly sensitive to the physical environment. To make the experiment independent from environmental influences, we create a shielded wireless testbed using pos. We conduct a measurement series on the PRRT protocol, a protocol designed with the requirements of NCS in mind. Our system monitors the channel properties at runtime, utilizing the PRRT protocol and cooperates with the control application to select the best available controller instance, i.e., gain scheduling. We further investigate four main levers for wireless control systems—sampling frequency, target delay, receiving window, and packet size—influencing the loss rates on wireless connections and identify potential bottlenecks. A detailed analysis of the WLAN network stack on the sender and receiver identifies medium access as the main bottleneck for wireless networks. Our fully automated evaluation procedure uses shielded boxes to create repeatable results in a well-defined environment for wireless measurements. We identify sampling frequency and target delay as the most critical impact factors on loss rate as long as receive windows are not chosen too narrow. Packet size only has a limited influence on loss rates.

Based on the previous measurements, we apply the resource model to analyze the potential bottlenecks for WLANs. All system interconnects and the CPU processing capabilities far exceed the required bandwidth and computing power. The only remaining bottleneck is the IEEE 802.11g standard itself that limits the achievable packet rates depending on the frame size of the transmitted traffic. We measure an additional impact of the WLAN implementation of our adapters that further decreases the achievable throughput. Utilizing this implementation-specific decrease and the theoretical throughput, we could successfully predict the maximum packet rates of IEEE 802.11g networks.

6.6 Author's Contributions

Section 6.1 is based on a publication by Sebastian Gallenmüller, Stephan Günther, Maurice Leclaire, Samuele Zoppi, Fabio Molinari, Richard Schöffauer, Wolfgang Kellerer, and Georg

Carle [107]. The author described the main ideas and concepts for the networking side of the benchmark.

Sections 6.1 and 6.2 further present NCSbench—a joint effort between Samuele Zoppi, Onur Ayan, Fabio Molinari, Zenit Music, Sebastian Gallenmüller, Georg Carle, and Wolfgang Kellerer [108]. The author contributed significantly to the implementation of NCSbench, the experiment platform itself, and the benchmark. In addition, the author performed measurements (platform A) and contributed significantly to the comparison of the platforms.

Section 6.3 is based on joint work between Sebastian Gallenmüller, René Glebke, Stephan Günther, Eric Hauser, Maurice Leclaire, Stefan Reif, Jan Rüth, Andreas Schmidt, Georg Carle, Thorsten Herfet, Wolfgang Schröder-Preikschat, and Klaus Wehrle [137]. The author supervised the thesis in which the PRRT measurements were performed and contributed his ideas and the measurement methodology. Further, significant contributions were made to the analysis and presentation of the data. The measurements presented in Figures 6.13 and 6.14 were created for this thesis. Based on the two figures, the analysis goes significantly beyond the investigation of the original work.

The model presented in Section 6.4 is novel, where the measurements and analysis were conducted by the author.

Chapter 7

Conclusion

This chapter summarizes the findings of this thesis on how to measure and model different packet processing systems. The continuing technical progress in the area of computer networks provides a fertile ground for new research questions. We want to end this chapter with a discussion of several open questions worth investigating.

7.1 Key Findings

The key findings are structured according to the research questions introduced in Section 1.1.

RQ1: How can we design and execute reproducible experiments for the investigation of packet processing systems? We presented our testbed with the testbed controller pos in Chapter 2. Following the pos experiment workflow, users have to create an experiment that is fully scripted and can be executed automatically. This kind of automation creates repeatable experiments, thereby reaching the first stage of reproducibility. In the case of shared testbed access, other researchers can reuse these scripts to replicate experiments—the second stage of reproducibility. Repeatability and replicability are an inherent consequence of pos experiment workflow, which we call replicability by design. We further demonstrated that the experiment results, together with the experiment and evaluation scripts, can be released with minor additional effort. This data provides other research groups with the necessary information to reproduce our experiments reaching the third stage, reproducibility [106].

The testbed and its controller were instrumental to the experiments presented in this thesis to different application domains. Chapter 5 contains experiments for high-performance packet processing applications based on off-the-shelf servers. In Chapter 6, we included shielded boxes into the testbed, which brings repeatable network experiments to the area of wireless networks.

RQ2: How can we create a measurement methodology to identify the main impact factors on packet processing performance? A tool that is central to the measurements of this thesis is MoonGen. We show that traffic properties, such as the inter-packet gap, can have a visible impact on the performance of packet processing systems. MoonGen offers several possibilities

to precisely determine the inter-packet gap according to the specification of the user. It further offers hardware timestamping capabilities for specific hardware, even timestamping of every received packet. These timestamps allow a detailed analysis of worst-case delays, as demonstrated in Chapter 5.

We further demonstrate that traditional architectures fail at bandwidths of 100G. Therefore, we present novel prototypes suitable for such high bandwidths. Both prototypes, FLOWer and Flowscope, leverage hardware offloading and specialized data structures to allow high-bandwidth measurements and analysis.

RQ3: How can we create a modeling framework to efficiently and adequately describe the behavior of packet processing systems in general? We identified different potential bottlenecks in the current hardware architectures of packet processing systems. Among these potential bottlenecks are the CPU performing the packet processing task, the maximum supported bandwidth of the NIC, or various internal system buses, such as PCIe. Depending on the amount and the quality of the traffic, only a subset of the identified components is responsible for the overall performance of the packet processing system. We created the resource model that utilizes these different bottlenecks to predict the overall performance of a system.

We demonstrated the application of the resource model to predict the performance of packet processing frameworks, an NFC framework, a software router, and an intrusion prevention system. The resource model requires the available bottleneck bandwidths and an approximation of the computational complexity of the processing task to predict the system capacity. Our resource model can be used to create packet processing systems that meet expected capacity and service levels. The modeling technique was also picked up by other research groups; Suksomboon et al. [150] extended the presented resource model to predict performance under the influence of cache contention.

RQ4: How can we characterize, analyze, and model high-performance packet processing systems? We demonstrated that, despite the simple two-node setup, many different investigations could be performed. This thesis presents an in-depth performance analysis of high-speed packet processing frameworks. The measurement methodology is further applied to an NFC framework based on the high-speed packet processing framework Snabb. In addition, we present MoonRoute, a high-performance software router, as an example of a high-performance packet processing application. Our measurements show that the performance of an adequately designed application scales almost linearly with CPU clock rate or cores. All measurements demonstrate that, utilizing these frameworks, performance can be improved to process bandwidths of 10 Gbit/s or more

The most sophisticated setup is applied to our investigation of an intrusion prevention system. There, we introduce a third measurement host, which timestamps every packet it observes. The third host allows a more detailed latency analysis compared with the original latency sampling process. Our measurements show that these high-speed processing frameworks are not only capable of increasing packet throughput, but are also suitable for designing low-latency applications.

RQ5: How can we characterize, analyze, and model wireless networked control systems? The thesis applies the measurement methodology to the application domain of wireless NCS. The conditions in this application environment are significantly different from the other investigated domains—bandwidths decrease and latencies increase by a factor of 1000, respectively. Control systems are typically resource constraint and wireless networks are susceptible to interference. All these factors required a revision of the entire measurement methodology.

We created our own platform and benchmark for NCS, which we designed with reproducibility and measurability in mind. Further, we upgraded our testbed with shielding capabilities to allow repeatable wireless network experiments. We demonstrate that the proposed resource model is also applicable to IEEE 802.11 WLAN networks. WLANs behave similar to wired networks, e.g., packet rates are more important than raw throughput rates. Our tests demonstrate that the main bottleneck of the IEEE 802.11g WLAN is the throughput limit contained in the network standard, which defines a complex medium access scheme. Another critical factor is the implementation of WLAN in hardware and software, which further limits the maximum achievable packet rates. System interconnects and the CPU do not present a relevant bottleneck as their capacity far exceeds the WLAN throughput limits.

7.2 Future Work

Our results show that we successfully established a methodology to perform reproducible network experiments. The current implementation of our testbeds has limitations, e.g., a residual state on a DuT that is not fully controllable via the testbed infrastructure, such as BIOS settings or firmware versions. However, these issues can be solved through additional development efforts and do not prevent a successful repetition of experiments in general.

This thesis shows that repeatable network experiments are possible even for highly sensitive wireless networks. We further present NCSbench, a reproducible benchmarking suite for NCS. Despite its similar focus, this thesis did not combine both approaches. Due to space constraints of our shielded boxes, we could not run the TWIPR inside the shielded environment. A larger shielded box would allow for a fully controlled wireless environment for the TWIPR. This controlled environment would allow the creation of interference reliably and repeatably to benchmark NCS even under challenging conditions. Additionally, a mechanism is needed to bring the robot into an upright position as the motors do not allow the robot to erect itself. Combining both approaches, the testbed and NCSbench, would allow replicable NCS measurements, a goal that we successfully achieved for wired measurements.

We demonstrated that the testbed could handle the bandwidths for 10G Ethernet; we also introduce tools capable of handling 100G traffic. However, with 400G Ethernet currently being rolled out, the bandwidths become too high to do complex packet processing on current off-the-shelf servers. We already showed a possible solution, the FLOWer approach [29], where we combine a switch with an off-the-shelf server. This switch can be programmed, which allows a partial offload of packet processing to a highly optimized, powerful switching ASIC. The load on the critical component of the server—the CPU—is minimized, allowing packet processing at 400G

and beyond. New network programming paradigms, such as P4, bring even more flexibility and processing power to the data plane. However, the CPU-based packet processing system allows highly complex algorithms that are not possible in the P4-based architectures. Therefore, we believe that future high-performance packet processing systems will rely on such a combination of systems to cope with ever-growing traffic. This thesis provides a foundation to measure, understand, and model current high-performance packet processing applications. However, additional effort will be required to upgrade the measurement capabilities to understand and, subsequently, model these novel combined packet processing systems.

Chapter A

Appendix

A.1 List of Acronyms

AP	Access point. Provides network access to end-user devices, often used in WLAN.
CAT	Cache allocation technology. Feature of Intel CPUs to partition shared cache among multiple CPU cores.
CBR	Constant bitrate. Rate which does not change over time. Constant bitrate requires a constant inter-packet gap.
CPS	Cyber-physical system. Physical system measured and controlled by a computer.
DDIO	Data Direct I/O. Technique to read/write from/into the last level cache instead of the main memory.
DMA	Direct Memory Access. Technique for I/O devices to read/write from/into the main memory bypassing the CPU.
DPDK	Data Plane Development Kit. Framework for creating high-performance packet processing applications.
DuT	Device under test. Subject of investigation in an experiment.
FFI	Foreign function interface. Interface to call functions written in another programming language.
KPI	Key performance indicator. Value to determine the performance of a system.
KVM	Kernel Virtual Machine. Virtualization solution that is part of the Linux kernel.
LLC	Last level cache. CPU cache between CPU and main memory, typically Level 3 cache.
MSE	Mean squared error. Average squared difference between a measured value and its estimation.
NCS	Networked control system. Control system controlled over a network connection.

NF	Network function. A packet processing task running in software.
NFC	Network function chain. Concatenation of network functions to perform complex packet processing tasks.
NIC	Network interface card. Network adapter typically used in a host or server.
OS	Operating system. Basic system software.
OvS	Open vSwitch. An open-source software switch.
pos	Plain orchestrating service. Testbed controller for designing and executing repeatable and replicable network experiments.
PRRT	Predictably reliable realtime transport. Transport layer protocol with configurable reliability and realtime properties.
QoC	Quality of control. Property to describe the quality of a control process.
QoS	Quality of service. Property to describe the quality of a service, typically an application.
QPI	Quick Path Interconnect. Intel CPU interconnect.
RSS	Receive Side Scaling. Distribution of received packets across a number of NIC queues to support multicore network applications.
SR-IOV	Single root IO virtualization. Technology to efficiently share PCIe devices, often used for VM IO.
TSC	Time stamp counter. Precise timer on modern x86 CPUs.
TWIPR	Two-wheeled inverted pendulum robot. Common experimental platform for control systems.
UPI	Ultra Path Interconnect. Intel CPU interconnect.

A.2 List of Figures

A.3 List of Tables

Bibliography

Work with author's contribution

[2] D. Raumer, S. Gallenmüller, F. Wohlfart, P. Emmerich, P. Werneck, and G. Carle, "Revisiting Benchmarking Methodology for Interconnect Devices", in *Proceedings of the 2016 Applied Networking Research Workshop, ANRW 2016, Berlin, Germany, July 16, 2016*, 2016, pp. 55–61. DOI: 10.1145/2959424.2959430. [Online]. Available: https://doi.org/10.1145/2959424.2959430.

[3] S. Gallenmüller, D. Scholz, F. Wohlfart, Q. Scheitle, P. Emmerich, and G. Carle, "High-Performance Packet Processing and Measurements", in *10th International Conference on Communication Systems & Networks, COMSNETS 2018, Bengaluru, India, January 3-7, 2018*, 2018, pp. 1–8. DOI: 10.1109/COMSNETS.2018.8328173. [Online]. Available: https://doi.org/10.1109/COMSNETS.2018.8328173.

[4] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5G QoS: Impact of Security Functions on Latency", in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*, IEEE, 2020, pp. 1–9. DOI: 10.1109/NOMS47738.2020.9110422. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110422.

[14] P. Emmerich, S. Gallenmüller, G. Antichi, A. W. Moore, and G. 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 18-19, 2017*, 2017, pp. 191–203. DOI: 10.1109/ANCS.2017.32. [Online]. Available: https://doi.org/10.1109/ANCS.2017.32.

[15] S. Gallenmüller, P. Emmerich, D. Raumer, and G. Carle, "MoonGen: Software Packet Generation for 10 Gbit and Beyond", in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, Oakland, CA, USA, May 4-6, 2015*, Poster, 2015.

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator", in *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, 2015, pp. 275–287. DOI: 10.1145/2815675.2815692. [Online]. Available: https://doi.org/10.1145/2815675.2815692.

[29] P. Emmerich, S. Gallenmüller, and G. Carle, "FLOWer - Device Benchmarking Beyond 100 Gbit/s", in *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*, 2016, pp. 109–116. DOI: 10.1109/IFIPNetworking.2016.7497198. [Online]. Available: https://doi.org/10.1109/IFIPNetworking.2016.7497198.

[33] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "FlowScope: Efficient Packet Capture and Storage in 100 Gbit/s Networks", in *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops, Stockholm, Sweden, June 12-16, 2017*, 2017, pp. 1–9. DOI: 10.

23919/IFIPNetworking.2017.8264852. [Online]. Available: https://doi.org/10.23919/IFIPNetworking.2017.8264852.

[38] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of Frameworks for High-Performance Packet IO", in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015*, 2015, pp. 29–38. DOI: 10.1109/ANCS.2015.7110118. [Online]. Available: https://doi.org/10.1109/ANCS.2015.7110118.

[39] S. Gallenmüller, "Comparison of Memory Mapping Techniques for High-Speed Packet Processing", Master's thesis, Technical University of Munich, 2014.

[77] ——, *SHEEP: Simulation algoritHms for Empirical Evaluation of Processor performance*, Last accessed: 2021-05-31, 2015. [Online]. Available: https://github.com/gallenmu/SHEEP.

[78] W. Hahn, B. Gajic, F. Wohlfart, D. Raumer, P. Emmerich, S. Gallenmüller, and G. Carle, "Feasibility of Compound Chained Network Functions for Flexible Packet Processing", in *International Workshop on 5G Enabling Technologies for the Internet of Things (GET-IoT) at the 23rd European Wireless (EW2017), Dresden, Germany, May 17-19, 2017*, 2017.

[80] S. Gallenmüller, P. Emmerich, R. Schönberger, D. Raumer, and G. Carle, "Building Fast but Flexible Software Routers", in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2017, Beijing, China, May 18-19, 2017*, 2017, pp. 101–102. DOI: 10.1109/ANCS.2017.21. [Online]. Available: https://doi.org/10.1109/ANCS.2017.21.

[81] P. Emmerich, S. Gallenmüller, R. Schönberger, D. Raumer, and G. Carle, "Architectures for Fast and Flexible Software Routers", Technical University of Munich, Garching near Munich, Germany, Tech. Rep., 2015. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonRoute_draft1.pdf.

[84] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis", *J. Network Syst. Manage.*, vol. 26, no. 2, pp. 314–338, 2018. DOI: 10.1007/s10922-017-9417-0. [Online]. Available: https://doi.org/10.1007/s10922-017-9417-0.

[87] P. Emmerich, D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T. M. Runge, S. Gallenmüller, and G. Carle, "Optimizing Latency and CPU Load in Packet Processing Systems", in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems, Chicago, IL, USA, July 26-29, 2015*, 2015, 6:1–6:8. DOI: 10.1109/SPECTS.2015.7285275. [Online]. Available: https://doi.org/10.1109/SPECTS.2015.7285275.

[106] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, *Reproducing Evaluation Results*, Last accessed: 2021-05-31, 2020. [Online]. Available: https://gallenmu.github.io/low-latency.

[107] S. Gallenmüller, S. M. Günther, M. Leclaire, S. Zoppi, F. Molinari, R. Schöffauer, W. Kellerer, and G. Carle, "Benchmarking Networked Control Systems", in *Workshop on Benchmarking Cyber-Physical Networks and Systems, Bench@CPSWeek 2018, Porto, Portugal, April 10, 2018*, 2018, pp. 7–12. DOI: 10.1109/CPSBench.2018.00008. [Online]. Available: https://doi.org/10.1109/CPSBench.2018.00008.

[108] S. Zoppi, O. Ayan, F. Molinari, Z. Music, S. Gallenmüller, G. Carle, and W. Kellerer, "NCSbench: Reproducible Benchmarking Platform for Networked Control Systems", in *IEEE 17th Annual Consumer Communications & Networking Conference, CCNC 2020, Las Vegas, NV, USA, January 10-13, 2020*, IEEE, 2020, pp. 1–9. DOI: 10.1109/CCNC46108.2020.9045199. [Online]. Available: https://doi.org/10.1109/CCNC46108.2020.9045199.

[112] ——, *NCSbench repository*, Last accessed: 2021-05-31, 2020. [Online]. Available: https://github.com/tum-lkn/NCSbench.

[135] Z. Music, F. Molinari, S. Gallenmüller, O. Ayan, S. Zoppi, W. Kellerer, G. Carle, T. Seel, and J. Raisch, "Design of a Networked Controller for a Two-Wheeled Inverted Pendulum Robot", *IFAC-PapersOnLine*, vol. 50, pp. 169–174, 20 2019. DOI: 10.1016/j.ifacol.2019.12.153. [Online]. Available: https://doi.org/10.1016/j.ifacol.2019.12.153.

[137] S. Gallenmüller, R. Glebke, S. M. Günther, E. Hauser, M. Leclaire, S. Reif, J. Rüth, A. Schmidt, G. Carle, T. Herfet, W. Schröder-Preikschat, and K. Wehrle, "Enabling Wireless Network Support for Gain Scheduled Control", in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, EdgeSys@EuroSys 2019, Dresden, Germany, March 25, 2019*, 2019, pp. 36–41. DOI: 10.1145/3301418.3313943. [Online]. Available: https://doi.org/10.1145/3301418.3313943.

References

[1] S. O. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices", *RFC*, vol. 2544, pp. 1–31, 1999. DOI: 10.17487/RFC2544. [Online]. Available: https://doi.org/10.17487/RFC2544.

[5] ACM, *Artifact Review and Badging*, Last accessed: 2021-05-31. [Online]. Available: http://www.acm.org/publications/policies/artifact-review-badging.

[6] Q. Scheitle, M. Wählisch, O. Gasser, T. C. Schmidt, and G. Carle, "Towards an Ecosystem for Reproducible Research in Computer Networking", in *Proceedings of the Reproducibility Workshop, Reproducibility@SIGCOMM 2017, Los Angeles, CA, USA, August 25, 2017*, ACM, 2017, pp. 5–8. DOI: 10.1145/3097766.3097768. [Online]. Available: https://doi.org/10.1145/3097766.3097768.

[7] N. Zilberman and A. W. Moore, "Thoughts about Artifact Badging", *Computer Communication Review*, vol. 50, no. 2, pp. 60–63, 2020. DOI: 10.1145/3402413.3402422. [Online]. Available: https://doi.org/10.1145/3402413.3402422.

[8] V. Bajpai, A. Brunström, A. Feldmann, W. Kellerer, A. Pras, H. Schulzrinne, G. Smaragdakis, M. Wählisch, and K. Wehrle, "The Dagstuhl Beginners Guide to Reproducibility for Experimental Networking Research", *Computer Communication Review*, vol. 49, no. 1, pp. 24–30, 2019. DOI: 10.1145/3314212.3314217. [Online]. Available: https://doi.org/10.1145/3314212.3314217.

[9] L. Nussbaum, "Testbeds Support for Reproducible Research", in *Proceedings of the Reproducibility Workshop, Reproducibility@SIGCOMM 2017, Los Angeles, CA, USA, August 25, 2017*, ACM, 2017, pp. 24–26. DOI: 10.1145/3097766.3097773. [Online]. Available: https://doi.org/10.1145/3097766.3097773.

[10] N. Zilberman, "An Artifact Evaluation of NDP", *Computer Communication Review*, vol. 50, no. 2, pp. 32–36, 2020. DOI: 10.1145/3402413.3402418. [Online]. Available: https://doi.org/10.1145/3402413.3402418.

[11] M. P. Grosvenor, M. Schwarzkopf, I. Gog, and A. W. Moore, "Jump the Queue to Lower Latency", *;login:*, vol. 40, no. 2, 2015. [Online]. Available: https://www.usenix.org/publications/login/apr15/grosvenor.

[12] Molex, *PXC3096 Datasheet*.

[13] O. S. Sella, A. W. Moore, and N. Zilberman, "FEC Killed The Cut-Through Switch", in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, NEAT@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, 2018, pp. 15–20. DOI: 10.1145/3229574.3229577. [Online]. Available: https://doi.org/10.1145/3229574.3229577.

[17] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua-An Extensible Extension Language", *Softw., Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996. DOI: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P. [Online]. Available: https://doi.org/10.1002/(SICI)1097-024X(199606)26:6%3C635::AID-SPE26%3E3.0.CO;2-P.

[18] M. Pall, *LuaJIT website*, Last accessed: 2021-05-31. [Online]. Available: https://luajit.org/luajit.html.

[19] *Intel Ethernet Controller X550 - Datasheet*, 333369-005, Rev. 2.3, Intel, Nov. 2018.

[20] *Intel 82599 10 GbE Controller - Datasheet*, 331520-005, Rev. 3.4, Intel, Nov. 2019.

[21] *Intel Ethernet Controller X710/ XXV710/XL710 Datasheet*, 332464-020, Rev. 3.65, Intel, Aug. 2019.

[22] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, G. A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: Open Source Network Tester", *IEEE Network*, vol. 28, no. 5, pp. 6–12, 2014. DOI: 10.1109/MNET.2014.6915433. [Online]. Available: https://doi.org/10.1109/MNET.2014.6915433.

[23] K. Wiles, *Pktgen-DPDK repository*, Last accessed: 2021-05-31. [Online]. Available: http://git.dpdk.org/apps/pktgen-dpdk/refs/.

[24] P. Emmerich, *MoonGen repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/emmericp/moongen.

[25] L. Rizzo, *netmap repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/luigirizzo/netmap.

[26] N. Bonelli, *PFQ repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/pfq/PFQ.

[27] ntop, *PF_RING ZC repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/ntop/PF_RING.

[28] L. Rizzo, "netmap: a novel framework for fast packet I/O", in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo.

[30] M. Ramanujam and N. Zilberman, "Towards a Highly Scalable Network Tester", in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, ANCS 2018, Ithaca, NY, USA, July 23-24, 2018*, 2018, pp. 154–155. DOI: 10.1145/3230718.3232104. [Online]. Available: https://doi.org/10.1145/3230718.3232104.

[31] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan, "A Power Benchmarking Framework for Network Devices", in *NETWORKING 2009, 8th International IFIP-TC 6 Networking Conference, Aachen, Germany, May 11-15, 2009. Proceedings*, 2009, pp. 795–808. DOI: 10.1007/978-3-642-01399-7_62. [Online]. Available: https://doi.org/10.1007/978-3-642-01399-7_62.

[32] J. Harrington, *TEST – THROUGHPUT ALCHEMY USING A SNAKE TOPOLOGY*, Last accessed: 2021-05-31, 2013. [Online]. Available: http://thenetworksherpa.com/test-throughput-alchemy-using-a-snake-topology/.

[34] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic", in *Proceedings of the 5th Internet Measurement Conference, IMC 2005, Berkeley, California, USA, October 19-21, 2005*, 2005, pp. 267–272. [Online]. Available: http://www.usenix.org/events/imc05/tech/kornexl.html.

[35] M. Egorushkin, *AtomicQueue - Scalability Benchmark*, Last accessed: 2021-05-31, 2019. [Online]. Available: https://max0x7ba.github.io/atomic_queue/html/benchmarks.html.

[36] H. Akaike, “A New Look at the Statistical Model Identification”, in *IEEE Transactions on Automatic Control*, IEEE, 1974, pp. 716–723.

[37] G. Schwarz, “Estimating the Dimension of a Model”, *The Annals of Statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[40] *Intel Data Direct I/O Technology (Intel DDIO): A Primer*, Rev. 1.0, Intel, Feb. 2012.

[41] *An Introduction to the Intel QuickPath Interconnect*, 320412-001US, Intel, Jan. 2009.

[42] D. Mulnix, *Intel Xeon Processor Scalable Family Technical Overview*, Last accessed: 2021-05-31. [Online]. Available: `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`.

[43] PCI-SIG, *Express Base Specification Revision 4.0*, 2017.

[44] JEDEC Solid State Technology Association, “JEDEC Standard: DDR3 SDRAM”, 2008.

[45] ——, “JEDEC Standard: DDR4 SDRAM”, 2012.

[46] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 248966-042b, Intel, Sep. 2019.

[47] DPDK Project, *DPDK repository*, Last accessed: 2021-05-31. [Online]. Available: `http://git.dpdk.org`.

[48] L. Rizzo and G. Lettieri, “VALE, a Switched Ethernet for Virtual Machines”, in *Conference on emerging Networking Experiments and Technologies, CoNEXT ’12, Nice, France - December 10 - 13, 2012*, 2012, pp. 61–72. DOI: `10.1145/2413176.2413185`. [Online]. Available: `https://doi.org/10.1145/2413176.2413185`.

[49] Linux Foundation, *Open vSwitch Release Notes*, Last accessed: 2021-05-31. [Online]. Available: `https://www.openvswitch.org/releases/NEWS-2.3.0.txt`.

[50] L. Rizzo, *netmap-ipfw repository*, Last accessed: 2021-05-31. [Online]. Available: `https://github.com/luigirizzo/netmap-ipfw`.

[51] F. Fusco and L. Deri, “High Speed Network Traffic Analysis with Commodity Multi-core Systems”, in *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, 2010, pp. 218–224. DOI: `10.1145/1879141.1879169`. [Online]. Available: `https://doi.org/10.1145/1879141.1879169`.

[52] R. Huggahalli, R. R. Iyer, and S. Tetrick, “Direct Cache Access for High Bandwidth Network I/O”, in *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*, 2005, pp. 50–59. DOI: `10.1109/ISCA.2005.23`. [Online]. Available: `https://doi.org/10.1109/ISCA.2005.23`.

[53] J. H. Salim, “When NAPI Comes to Town”, in *Linux 2005 Conference, 2005, Swansea, Wales, United Kingdom, 4-7 August, 2005*, 2005.

[54] M. Dobrescu, K. J. Argyraki, and S. Ratnasamy, “Toward Predictable Performance in Software Packet-Processing Platforms”, in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 141–154. [Online]. Available: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu`.

[55] R. Bolla and R. Bruschi, “Linux Software Router: Data Plane Optimization and Performance Evaluation”, *JNW*, vol. 2, no. 3, pp. 6–17, 2007, [Online, alternative link]. Available: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.844&rep=rep1&type=pdf`. DOI: `10.4304/jnw.2.3.6-17`. [Online]. Available: `https://doi.org/10.4304/jnw.2.3.6-17`.

[56] J. L. García-Dorado, F. Mata, J. Ramos, P. M. S. del Río, V. Moreno, and J. Aracil, “High-Performance Network Traffic Processing Systems Using Commodity Hardware”, in *Data Traffic Monitoring and Analysis - From Measurement, Classification, and Anomaly Detection to Quality of Experience*, 2013, pp. 3–27. DOI: `10.1007/978-3-642-36784-7_1`. [Online]. Available: `https://doi.org/10.1007/978-3-642-36784-7_1`.

[57] L. Deri, "nCap: Wire-speed Packet Capture and Transmission", in *Third IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, E2EMON 2005, 15th May 2005, Nice, France*, 2005, pp. 47–55. DOI: 10.1109/E2EMON.2005.1564468. [Online]. Available: https://doi.org/10.1109/E2EMON.2005.1564468.

[58] The FreeBSD Project, "FreeBSD 10.0-RELEASE Release Notes", Last accessed: 2021-05-31, 2014. [Online]. Available: https://www.freebsd.org/releases/10.0R/relnotes.html.

[59] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing", in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015*, 2015, pp. 5–16. DOI: 10.1109/ANCS.2015.7110116. [Online]. Available: https://doi.org/10.1109/ANCS.2015.7110116.

[60] ntop, *PF_RING API*, Last accessed: 2021-05-31. [Online]. Available: http://www.ntop.org/guides/pf_ring_api/pfring__zc_8h.html.

[61] ——, *ntop website*, Last accessed: 2021-05-31. [Online]. Available: https://ntop.org.

[62] DPDK Project, *DPDK Programmer's Guide*, Last accessed: 2021-05-31. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/.

[63] J. Corbet, *UIO: user-space drivers*, Last accessed: 2021-05-31, 2007. [Online]. Available: http://lwn.net/Articles/232575/.

[64] M. Sune, A. Koepsel, V. Alvarez, and T. Jungel, *xdpd repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/bisdn/xdpd.

[65] S. Han, K. Jang, K. Park, and S. B. Moon, "PacketShader: A GPU-Accelerated Software Router", in *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30-September 3, 2010*, 2010, pp. 195–206. DOI: 10.1145/1851182.1851207. [Online]. Available: https://doi.org/10.1145/1851182.1851207.

[66] S. Han, K. Jang, S. Huh, and J. Kim, *Packet-IO-Engine repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/ANLAB-KAIST/Packet-IO-Engine.

[67] N. Bonelli, A. D. Pietro, S. Giordano, and G. Procissi, "On Multi-gigabit Packet Capturing with Multi-core Commodity Hardware", in *Passive and Active Measurement - 13th International Conference, PAM 2012, Vienna, Austria, March 12-14, 2012. Proceedings*, 2012, pp. 64–73. DOI: 10.1007/978-3-642-28537-0_7. [Online]. Available: https://doi.org/10.1007/978-3-642-28537-0_7.

[68] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, "A Purely Functional Approach to Packet Processing", in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems, ANCS 2014, Los Angeles, CA, USA, October 20-21, 2014*, 2014, pp. 219–230. DOI: 10.1145/2658260.2658269. [Online]. Available: https://doi.org/10.1145/2658260.2658269.

[69] SnabbCo, *Snabb*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/snabbco/snabb.

[70] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK", in *Second European Workshop on Software Defined Networks, EWSDN 2013, Berlin, Germany, October 10-11, 2013*, 2013, pp. 62–67. DOI: 10.1109/EWSDN.2013.17. [Online]. Available: https://doi.org/10.1109/EWSDN.2013.17.

[71] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems", in *Fourteenth International Conference on Networks (ICN 2015), Barcelona, Spain, April 19-24, 2015*, 2015. [Online]. Available: http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ICN2015.pdf.

[72] L. Angrisani, G. Ventre, L. Peluso, and A. Tedesco, "Measurement of Processing and Queuing Delays Introduced by an Open-Source Router in a Single-Hop Network", *IEEE Trans. Instrumentation and Measurement*, vol. 55, no. 4, pp. 1065–1076, 2006. DOI: 10.1109/TIM.2006.876542. [Online]. Available: https://doi.org/10.1109/TIM.2006.876542.

[73] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation", in *Passive and Active Measurement - 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012. Proceedings*, 2012, pp. 85–95. DOI: 10.1007/978-3-642-28537-0_9. [Online]. Available: https://doi.org/10.1007/978-3-642-28537-0_9.

[74] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network", *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009. DOI: 10.1007/s10766-009-0109-6. [Online]. Available: https://doi.org/10.1007/s10766-009-0109-6.

[75] L. Braun, C. Diekmann, N. Kammenhuber, and G. Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware", in *IFIP Networking Conference, 2013, Brooklyn, New York, USA, 22-24 May, 2013*, 2013, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/6663536/.

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

[79] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. López, F. J. R. Salguero, F. Ruhl, and P. Sen, "Network Functions Virtualisation", in *SDN and OpenFlow World Congress, Darmstadt, Germany, October 22-24, 2012*, 2012. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[82] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds", in *Proceedings IEEE INFOCOM '98, The Conference on Computer Communications, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, Gateway to the 21st Century, San Francisco, CA, USA, March 29 - April 2, 1998*, IEEE Computer Society, 1998, pp. 1240–1247. DOI: 10.1109/INFCOM.1998.662938. [Online]. Available: https://doi.org/10.1109/INFCOM.1998.662938.

[83] ITU, *Report ITU-R M.2410-0 (11/2017) Minimum requirements related to technical performance for IMT-2020 radio interface(s)*, Last accessed: 2021-05-31. [Online]. Available: https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2410-2017-PDF-E.pdf.

[85] 3GPP, *3GPP TS 22.104 V17.2.0 (2019-12)*, Last accessed: 2021-05-31, Dec. 2019. [Online]. Available: https://www.3gpp.org/ftp/Specs/archive/22_series/22.104.

[86] T. Yoshizawa, S. B. M. Baskaran, and A. Kunz, "Overview of 5G URLLC System and Security Aspects in 3GPP", in *2019 IEEE Conference on Standards for Communications and Networking, CSCN 2019, Granada, Spain, October 28-30, 2019*, IEEE, 2019, pp. 1–5. DOI: 10.1109/CSCN.2019.8931376. [Online]. Available: https://doi.org/10.1109/CSCN.2019.8931376.

[88] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel", in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, 2018, pp. 54–66. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[89] Napatech, *Snort DAQ repository*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/napatech/daq_dpdk_multiqueue.

[90] M. Beierl, *Nfv-kvm-tuning*, Last accessed: 2021-05-31. [Online]. Available: https://wiki.opnfv.org/pages/viewpage.action?pageId=2926179.

[91] J. Mario and J. Eder, *Low Latency Performance Tuning for Red Hat Enterprise Linux 7*, Version 2.1, Last accessed: 2021-05-31, Nov. 2017. [Online]. Available: https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf.

[92] AMD, *Performance Tuning Guidelines for Low Latency Response on AMD EPYC-Based Servers*, Last accessed: 2021-05-31. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/PerformanceTuningGuidelinesforLowLatencyResponse.pdf.

[93] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers", in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, 2014, pp. 406–418. DOI: 10.1109/MICRO.2014.53. [Online]. Available: https://doi.org/10.1109/MICRO.2014.53.

[94] R. Schöne, D. Molka, and M. Werner, "Wake-up latencies for processor idle states on current x86 processors", *Computer Science - R&D*, vol. 30, no. 2, pp. 219–227, 2015. DOI: 10.1007/s00450-014-0270-z. [Online]. Available: https://doi.org/10.1007/s00450-014-0270-z.

[95] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family", in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 657–668. DOI: 10.1109/HPCA.2016.7446102. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446102.

[96] P. McKenney, *A realtime preemption overview*, Last accessed: 2021-05-31. [Online]. Available: https://lwn.net/Articles/146861/.

[97] G. Lettieri, V. Maffione, and L. Rizzo, "A Survey of Fast Packet I/O Technologies for Network Function Virtualization", in *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, 2017, pp. 579–590. DOI: 10.1007/978-3-319-67630-2_40. [Online]. Available: https://doi.org/10.1007/978-3-319-67630-2_40.

[98] X. Xu and B. Davda, "SRVM: Hypervisor Support for Live Migration with Passthrough SR-IOV Network Devices", in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Atlanta, GA, USA, April 2-3, 2016*, 2016, pp. 65–77. DOI: 10.1145/2892242.2892256. [Online]. Available: https://doi.org/10.1145/2892242.2892256.

[99] Z. Xiang, F. Gabriel, E. Urbano, G. T. Nguyen, M. Reisslein, and F. H. P. Fitzek, "Reducing Latency in Virtual Machines: Enabling Tactile Internet for Human-Machine Co-Working", *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 1098–1116, 2019. DOI: 10.1109/JSAC.2019.2906788. [Online]. Available: https://doi.org/10.1109/JSAC.2019.2906788.

[100] N. Zilberman, M. P. Grosvenor, D. A. Popescu, N. M. Bojan, G. Antichi, M. Wójcik, and A. W. Moore, "Where Has My Time Gone?", in *Passive and Active Measurement - 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings*, 2017, pp. 201–214. DOI: 10.1007/978-3-319-54328-4_15. [Online]. Available: https://doi.org/10.1007/978-3-319-54328-4_15.

[101] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, no VM Exits! (Almost)", *CoRR*, vol. abs/1705.06932, 2017. arXiv: 1705.06932. [Online]. Available: http://arxiv.org/abs/1705.06932.

[102] R. Kaiser and S. Wagner, "Evolution of the PikeOS Microkernel", in *First International Workshop on Microkernels for Embedded Systems*, vol. 50, Jan. 2007.

[103] Cisco Inc., *Snort*, Last accessed: 2021-05-31. [Online]. Available: https://github.com/snort3/snort3.

[104] Snort, *Snort3 community ruleset*, Last accessed: 2021-05-31. [Online]. Available: https://www.snort.org/downloads/#rule-downloads.

[105] S. Bauer, D. Raumer, P. Emmerich, and G. Carle, "Intra-Node Resource Isolation for SFC with SR-IOV", in *7th IEEE International Conference on Cloud Networking, CloudNet 2018, Tokyo, Japan, October 22-24, 2018*, 2018, pp. 1–6. DOI: 10.1109/CloudNet.2018.8549547. [Online]. Available: https://doi.org/10.1109/CloudNet.2018.8549547.

[109] K. Kim and P. R. Kumar, "Cyber-Physical Systems: A Perspective at the Centennial", *Proceedings of the IEEE*, vol. 100, no. Centennial-Issue, pp. 1287–1308, 2012. DOI: 10.1109/JPROC.2012.2189792. [Online]. Available: https://doi.org/10.1109/JPROC.2012.2189792.

[110] R. A. Gupta and M. Chow, "Networked Control System: Overview and Research Trends", *IEEE Trans. Industrial Electronics*, vol. 57, no. 7, pp. 2527–2535, 2010. DOI: 10.1109/TIE.2009.2035462. [Online]. Available: https://doi.org/10.1109/TIE.2009.2035462.

[111] L. Zhang, H. Gao, and O. Kaynak, "Network-Induced Constraints in Networked Control Systems—A Survey", *IEEE Trans. Industrial Informatics*, vol. 9, no. 1, pp. 403–416, 2013. DOI: 10.1109/TII.2012.2219540. [Online]. Available: https://doi.org/10.1109/TII.2012.2219540.

[113] L. Zhang, Y. Shi, T. Chen, and B. Huang, "A New Method for Stabilization of Networked Control Systems With Random Delays", *IEEE Trans. Automat. Contr.*, vol. 50, no. 8, pp. 1177–1181, 2005. DOI: 10.1109/TAC.2005.852550. [Online]. Available: https://doi.org/10.1109/TAC.2005.852550.

[114] X. Zhang, Q. Han, and X. Yu, "Survey on Recent Advances in Networked Control Systems", *IEEE Trans. Industrial Informatics*, vol. 12, no. 5, pp. 1740–1752, 2016. DOI: 10.1109/TII.2015.2506545. [Online]. Available: https://doi.org/10.1109/TII.2015.2506545.

[115] C. Lu, A. Saifullah, B. Li, M. Sha, H. González, D. Gunatilaka, C. Wu, L. Nie, and Y. Chen, "Real-Time Wireless Sensor-Actuator Networks for Industrial Cyber-Physical Systems", *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1013–1024, 2016. DOI: 10.1109/JPROC.2015.2497161. [Online]. Available: https://doi.org/10.1109/JPROC.2015.2497161.

[116] W. Zhang, M. S. Branicky, and S. M. Philips, "Stability of Networked Control Systems", *IEEE Control Systems Magazine*, vol. 21, no. 1, pp. 88–99, 2001. DOI: 10.1109/37.898794. [Online]. Available: https://doi.org/10.1109/37.898794.

[117] A. Chamaken and L. Litz, "Joint Design of Control and Communication in Wireless Networked Control Systems: A Case Study", 2010, pp. 1835–1840. DOI: 10.1109/ACC.2010.5531426. [Online]. Available: https://doi.org/10.1109/ACC.2010.5531426.

[118] P. A. Kawka and A. G. Alleyne, "Stability and Feedback Control of Wireless Networked Systems", 2005, pp. 2953–2959. DOI: 10.1109/ACC.2005.1470423. [Online]. Available: https://doi.org/10.1109/ACC.2005.1470423.

[119] J. Eker, A. Cervin, and A. Hörjel, "Distributed Wireless Control Using Bluetooth", *IFAC Proceedings Volumes*, vol. 34, no. 22, pp. 360–365, 2001. DOI: 10.1016/S1474-6670(17)32965-8. [Online]. Available: https://doi.org/10.1016/S1474-6670(17)32965-8.

[120] C. Bachhuber, S. Conrady, M. Schütz, and E. G. Steinbach, "A Testbed for Vision-Based Networked Control Systems", in *Computer Vision Systems - 11th International Conference, ICVS 2017, Shenzhen, China, July 10-13, 2017, Revised Selected Papers*, 2017, pp. 26–36. DOI: 10.1007/978-3-319-68345-4_3. [Online]. Available: https://doi.org/10.1007/978-3-319-68345-4_3.

[121] D. Baumann, F. Mager, H. Singh, M. Zimmerling, and S. Trimpe, "Evaluating Low-Power Wireless Cyber-Physical Systems", in *Workshop on Benchmarking Cyber-Physical Networks and Systems, Bench@CPSWeek 2018, Porto, Portugal, April 10, 2018*, 2018, pp. 13–18. DOI: 10.1109/CPSBench.2018.00009. [Online]. Available: https://doi.org/10.1109/CPSBench.2018.00009.

[122] F. Mager, D. Baumann, R. Jacob, L. Thiele, S. Trimpe, and M. Zimmerling, "Feedback Control Goes Wireless: Guaranteed Stability over Low-power Multi-hop Networks", in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019*, 2019, pp. 97–108. DOI: 10.1145/3302509.3311046. [Online]. Available: https://doi.org/10.1145/3302509.3311046.

[123] N. J. Ploplys, P. A. Kawka, and A. G. Alleyne, "Closed-Loop Control over Wireless Networks", *IEEE Control Systems Magazine*, vol. 24, no. 3, pp. 58–71, 2004. DOI: 10.1109/MCS.2004.1299533. [Online]. Available: https://doi.org/10.1109/MCS.2004.1299533.

[124] C. Peng, D. Yue, and M. Fei, "A Higher Energy-Efficient Sampling Scheme for Networked Control Systems over IEEE 802.15.4 Wireless Networks", *IEEE Trans. Industrial Informatics*, vol. 12, no. 5, pp. 1766–1774, 2016. DOI: 10.1109/TII.2015.2481821. [Online]. Available: https://doi.org/10.1109/TII.2015.2481821.

[125] S. Nethi, M. Pohjola, L. Eriksson, and R. Jäntti, "Platform for Emulating Networked Control Systems in Laboratory Environments", in *2007 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007), 18-21 June 2007, Helsinki, Finland, Proceedings*, 2007, pp. 1–8. DOI: 10.1109/WOWMOM.2007.4351727. [Online]. Available: https://doi.org/10.1109/WOWMOM.2007.4351727.

[126] L. Wu and G. Kaiser, "FARE: A Framework for Benchmarking Reliability of Cyber-Physical Systems", in *IEEE Long Island Systems, Applications and Technology Conference 2013, May 3, Farmingdale, NY, USA*, 2013. DOI: 10.1109/LISAT.2013.6578226. [Online]. Available: https://doi.org/10.1109/LISAT.2013.6578226.

[127] S. X. Ding, P. Zhang, S. Yin, and E. L. Ding, "An Integrated Design Framework of Fault-Tolerant Wireless Networked Control Systems for Industrial Automatic Control Applications", *IEEE Trans. Industrial Informatics*, vol. 9, no. 1, pp. 462–471, 2013. DOI: 10.1109/TII.2012.2214390. [Online]. Available: https://doi.org/10.1109/TII.2012.2214390.

[128] C. A. Boano, S. Duquennoy, A. Förster, O. Gnawali, R. Jacob, H. Kim, O. Landsiedel, R. Marfievici, L. Mottola, G. P. Picco, X. Vilajosana, T. Watteyne, and M. Zimmerling, "IoTBench: Towards a Benchmark for Low-Power Wireless Networking", in *Workshop on Benchmarking Cyber-Physical Networks and Systems, Bench@CPSWeek 2018, Porto, Portugal, April 10, 2018*, 2018, pp. 36–41. DOI: 10.1109/CPSBench.2018.00013. [Online]. Available: https://doi.org/10.1109/CPSBench.2018.00013.

[129] T. Niemueller, G. Lakemeyer, S. Reuter, S. Jeschke, and A. Ferrein, "Benchmarking of Cyber-Physical Systems in Industrial Robotics: The RoboCup Logistics League as a CPS Benchmark Blueprint", pp. 193–207, 2016. DOI: 10.1016/B978-0-12-803801-7.00013-4. [Online]. Available: https://doi.org/10.1016/B978-0-12-803801-7.00013-4.

[130] S. M. Günther, M. Leclaire, J. Michaelis, and G. Carle, "Analysis of Injection Capabilities and Media Access of IEEE 802.11 Hardware in Monitor Mode", in *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, 2014, pp. 1–9. DOI:

10.1109/NOMS.2014.6838262. [Online]. Available: https://doi.org/10.1109/NOMS.2014.6838262.

[131] D. Vassis, G. Kormentzas, A. N. Rouskas, and I. Maglogiannis, "The IEEE 802.11g Standard for High Data Rate WLANs", *IEEE Network*, vol. 19, no. 3, pp. 21–26, 2005. DOI: 10.1109/MNET.2005.1453395. [Online]. Available: https://doi.org/10.1109/MNET.2005.1453395.

[132] B. Wittenmark, J. Nilsson, and M. Torngren, "Timing Problems in Real-time Control Systems", 1995, pp. 2000–2004. DOI: 10.1109/ACC.1995.531240. [Online]. Available: https://doi.org/10.1109/ACC.1995.531240.

[133] C. Liu, F. Chen, J. Zhu, Z. Zhang, C. Zhang, C. Zhao, and T. Wang, "Characteristic, Architecture, Technology, and Design Methodology of Cyber-Physical Systems", in *International Conference on Industrial IoT Technologies and Applications*, 2017. DOI: 10.1007/978-3-319-60753-5_25. [Online]. Available: https://doi.org/10.1007/978-3-319-60753-5_25.

[134] Y. H. Kim, S. Kim, and Y. K. Kwak, "Dynamic Analysis of a Nonholonomic Two-Wheeled Inverted Pendulum Robot", *Journal of Intelligent and Robotic Systems*, vol. 44, no. 1, pp. 25–46, 2005. DOI: 10.1007/s10846-005-9022-4. [Online]. Available: https://doi.org/10.1007/s10846-005-9022-4.

[136] LEGO Group, *LEGO MINDSTORMS Education EV3 Core Set*, Last accessed: 2021-05-31. [Online]. Available: https://education.lego.com/en-us/products/lego-mindstorms-education-ev3-core-set-/5003400.

[138] M. Satyanarayanan, "The Emergence of Edge Computing", *IEEE Computer*, vol. 50, no. 1, pp. 30–39, 2017. DOI: 10.1109/MC.2017.9. [Online]. Available: https://doi.org/10.1109/MC.2017.9.

[139] D. J. Leith and W. E. Leithead, "Survey of Gain-Scheduling Analysis & Design", *International Journal of Control*, vol. 73, no. 11, pp. 1001–1025, 2000. DOI: 10.1080/002071700411304. [Online]. Available: https://doi.org/10.1080/002071700411304.

[140] R. Costa, P. Portugal, F. Vasques, C. Montez, and R. Moraes, "Limitations of the IEEE 802.11 DCF, PCF, EDCA and HCCA to handle real-time traffic", in *13th IEEE International Conference on Industrial Informatics, INDIN 2015, Cambridge, United Kingdom, July 22-24, 2015*, 2015, pp. 931–936. DOI: 10.1109/INDIN.2015.7281860. [Online]. Available: https://doi.org/10.1109/INDIN.2015.7281860.

[141] K. Nakashima, T. Matsuda, M. Nagahara, and T. Takine, "Cross-Layer Design of an LQG Controller in Multihop TDMA-Based Wireless Networked Control Systems", in *28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2017, Montreal, QC, Canada, October 8-13, 2017*, 2017, pp. 1–7. DOI: 10.1109/PIMRC.2017.8292181. [Online]. Available: https://doi.org/10.1109/PIMRC.2017.8292181.

[142] G. Nikolakopoulos, A. Panousopoulou, A. Tzes, and J. Lygeros, "Multi-hopping Induced Gain Scheduling for Wireless Networked Controlled Systems", *Asian Journal of Control*, vol. 9, no. 4, pp. 450–457, 2007. DOI: 10.1111/j.1934-6093.2007.tb00433.x. [Online]. Available: https://doi.org/10.1111/j.1934-6093.2007.tb00433.x.

[143] F. Xia, L. Ma, C. Peng, Y. Sun, and J. Dong, "Cross-Layer Adaptive Feedback Scheduling of Wireless Control Systems", *Sensors*, vol. 8, no. 7, pp. 4265–4281, 2008. DOI: 10.3390/s8074265. [Online]. Available: https://doi.org/10.3390/s8074265.

[144] A. Schmidt and S. Reif, *PRRT repository*, Last accessed: 2021-05-31. [Online]. Available: https://prrt.larn.systems.

[145] S. Reif, A. Schmidt, T. Hönig, T. Herfet, and W. Schröder-Preikschat, "ΔELTA: Differential Energy-Efficiency, Latency, and Timing Analysis for Real-Time Networks", *SIGBED Review*,

vol. 16, no. 1, pp. 33–38, 2019. DOI: 10.1145/3314206.3314211. [Online]. Available: https://doi.org/10.1145/3314206.3314211.

[146] J. Gettys and K. M. Nichols, "Bufferbloat: Dark Buffers in the Internet", *Commun. ACM*, vol. 55, no. 1, pp. 57–65, 2012. DOI: 10.1145/2063176.2063196. [Online]. Available: https://doi.org/10.1145/2063176.2063196.

[147] Y. Cheng, N. Cardwell, S. H. Yeganeh, and V. Jacobson, "Delivery Rate Estimation draft-cheng-iccrg-delivery-rate-estimation-00", pp. 1–15, 2017.

[148] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control", *ACM Queue*, vol. 14, no. 5, pp. 20–53, 2016. DOI: 10.1145/3012426.3022184. [Online]. Available: http://doi.acm.org/10.1145/3012426.3022184.

[149] J. L. Bordim, A. V. Barbosa, M. F. Caetano, and P. S. Barreto, "IEEE802.11b/g Standard: Theoretical Maximum Throughput", in *First International Conference on Networking and Computing, ICNC 2010, Higashi Hiroshima, Japan, November 17-19, 2010. Proceedings*, IEEE Computer Society, 2010, pp. 197–201. DOI: 10.1109/IC-NC.2010.40. [Online]. Available: https://doi.org/10.1109/IC-NC.2010.40.

[150] K. Suksomboon, M. Fukushima, S. Okamoto, and M. Hayashi, "A Dilated-CPU-Consumption-Based Performance Prediction for Multi-Core Software Routers", in *IEEE NetSoft Conference and Workshops, NetSoft 2016, Seoul, South Korea, June 6-10, 2016*, IEEE, 2016, pp. 193–201. DOI: 10.1109/NETSOFT.2016.7502413. [Online]. Available: https://doi.org/10.1109/NETSOFT.2016.7502413.