Network Architectures
and Services
NET 2022-04-1

Habilitation Thesis

Data-Driven Network Architectures and Protocols

Fabien Clément Geyer

Department of Informatics

Technische Universität München

TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

Data-Driven Network Architectures and Protocols

Fabien Clément Geyer

Habilitationsschrift

Fachmentorat:
Univ.-Prof. Dr. Tobias Nipkov (Vorsitzender)
Univ.-Prof. Dr.-Ing. Georg Carle
Prof. Dario Rossi, Ph.D. (Telecom ParisTech, France)

Die Habilitationsschrift wurde am 03.05.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.04.2022 angenommen.

Cataloging-in-Publication Data
Fabien Clément Geyer
Data-Driven Network Architectures and Protocols
Habilitationsschrift, June 2022
Network Architectures and Services, Department of Computer Science
Technische Universität München

ISBN 978-3-937201-74-0
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)
DOI 10.2313/NET-2022-04-1
Network Architectures and Services NET-2022-04-1
Series Editor: Georg Carle, Technische Universität München, Germany
© 2022, Technische Universität München, Germany

ABSTRACT

Network automation has long been a goal in the networking research community, with promises such as avoiding misconfiguration, automatically optimizing network performance, or dynamically reacting to changes in demand or network failures. In this Habilitation thesis we investigate data-driven methods for addressing such networking challenges. We present both theoretical results based on mathematical frameworks, as well as more practical results based on real implementations and measurements on testbeds.

The main contribution of this research work is a data-driven approach based on graph transformation and graph neural networks (GNNs). This approach stems from our finding that various challenges found in networking research can be modeled as graphs, a key data structure for representing network topologies, their properties and their configuration. Using this natural way of representing data and by adding expert knowledge, we build on the recent works from the machine learning (ML) community on GNNs to make efficient and accurate predictions about the problem to be solved.

We demonstrate in this Habilitation thesis that this approach can be applied to a variety of networking challenges, from formal analysis of networks, to design of network protocols. We first investigate network calculus (NC), a mathematical framework for computing bounds on end-to-end latencies. We propose the first applications of ML and GNN to NC analysis. One finding of our research is to illustrate that getting the tightest bound from NC often requires expert knowledge about the methods applied, as well as large computational costs. With our work, we alleviate those needs by using a data-driven approach and contribute two novel NC approaches based on GNNs: DeepTMA and DeepFP. We show that ML has its place inside NC, opening up the door to formally-verified tight bounds at a low computational cost.

We also demonstrate other applications of GNNs to networking challenges in this Habilitation thesis. We contribute one of the first works applying GNNs to the performance evaluation of elastic and inelastic flows, predicting flow bandwidth and packet latency. We show that GNNs can provide accurate and efficient predictions of flow performance, providing a fast tool for what-if analysis. We also contribute a novel method taking advantage of the message passing principle of GNNs for building network protocols. We illustrate that routing protocols can easily be trained for, requiring no expert knowledge on network protocol design. Finally, we contribute an application of GNNs to the analysis of Multiprotocol Label Switching (MPLS) networks and configurations. We show that our graph transformation is able to handle more advanced concepts such as MPLS forwarding rules and configuration.

Finally, we explore how lessons learned from data-driven approaches and their models can be included in the networks themselves. We focus on P4, a promising solution for including such advanced functionalities in the dataplane on various hardware platforms, as well as Vector Packet Processing (VPP), a software router based on recent developments for efficient software packet processing. We contribute an evaluation of P4 in an industrial context, showing that P4 could be used for implementing industrial protocols. We also show that P4 can easily be extended to include advanced functionalities such as secure and fast hash functions. Our performance evaluations showed promising results, achieving 10 Gbit/s line-rate in some cases. With VPP we also show that ML can be directly included in the dataplane to provide advanced online optimization of the packet processing pipeline.

ACKNOWLEDGMENTS

I'm grateful for the support and encouragements I received from certain people without whom this Habilitation work would not have been possible.

First of all, I would like to thank Prof. Georg Carle for mentoring me during this Habilitation and the various advices he gave me along the way since I started doing research. I would also like to thank Prof. Dario Rossi for his feedback on my work and being my second assessor and Prof. Tobias Nipkov for chairing the examination committee.

I would also like to say a big thank you to all my co-authors, which made our joint research a success. A special thanks goes to Steffen Bondorf for our fruitful collaborations and joint research. I'm also thankful to my former colleagues at TUM, especially Benedikt, Dominik, and Sebastian, and my current and past colleagues at Airbus.

Finalement, mon plus remerciement va à mes parents et ma famille, pour leur support, leur patience et leurs encouragements pendant ces longues années loin de la maison. Un grand merci à vous !

Munich, June 8, 2022

Fabien Geyer

CONTENTS

1. Introduction . . . 1
1.1 Research overview in scope of the Habilitation . . . 1
1.2 Research objectives and expected benefits . . . 3
1.3 Contributions and document structure . . . 4
1.4 Remarks about this document . . . 6

2. Background . . . 7
2.1 Deterministic network calculus . . . 7
2.2 Graph neural networks . . . 10
2.3 P4: An approach for efficient packet processing devices . . . 12
2.4 Data Driven Methods for Networking . . . 13

3. Tight and efficient bounds in network calculus with fast heuristics . . . 17
3.1 Introduction . . . 17
3.2 Related work . . . 18
3.3 Extending network calculus analyses for tighter bounds . . . 20
3.4 Graph-based deep learning for speeding up formal verification . . . 25
3.5 Conclusion on network calculus improvements . . . 36

4. Application of graph-based deep learning methods for computer networks . . . 37
4.1 Introduction . . . 37
4.2 Related work . . . 37
4.3 Graph models for performance evaluation with DeepComNet . . . 39
4.4 Graph models for generating network protocols . . . 42
4.5 Graph models for reasoning about network protocols . . . 45
4.6 Conclusion on machine learning methods for computer networks . . . 49

5. Hardware and software for efficient packet processing . . . 51
5.1 Introduction . . . 51
5.2 Related work . . . 52
5.3 Modular packet processing for fast prototyping . . . 53
5.4 Advanced secure hashes for in-network packet processing . . . 56
5.5 Adaptive ML-based batching for fast software routers . . . 59
5.6 Conclusion on efficient packet processing . . . 62

6. Conclusion . . . 63
6.1 Summary and conclusions . . . 63
6.2 Future research directions . . . 65

Appendix 67

A. Publications . . . 69
A.1 Tight and efficient bounds in network calculus with fast heuristics . . . 70
A.2 Application of graph-based deep learning methods for computer networks . . . 156
A.3 Hardware and software for efficient packet processing . . . 199
A.4 List of publications . . . 235

B. List of research artifacts . . . 239

C. Glossary . . . 241

Bibliography . . . 243

1. INTRODUCTION

1.1 Research overview in scope of the Habilitation 1
1.2 Research objectives and expected benefits 3
1.3 Contributions and document structure 4
1.4 Remarks about this document 6

1.1 Research overview in scope of the Habilitation

Building upon recent advances in self-driving cars and similar trends towards more automation in other domains, Feamster and Rexford [54] recently proposed the concept of *self-driving network*. The idea of a network with automation stems from the growing challenges faced by network design and management, which need to cope and optimize for different goals such as performance, availability, resilience to attack, ubiquitous access, and scale. The current approaches to those different goals is to develop locally-optimized solutions, namely solutions which are only applicable for a specific use-case and which cannot be easily ported to other use-cases.

The proposed approach for a self-driving network from [54] is to have a method which should take as input a high-level goal related to performance or security for example and jointly derive *(i)* the metrics and measurements that the network should collect; *(ii)* the inferences that should be performed; *(iii)* the decisions that the network should ultimately execute. There are currently no general solution for implementing such a self-driving network.

Machine learning (ML) has often been cited as a central solution to self-driving networks. In this work, we propose to use graph neural networks (GNNs) – a specific neural network (NN) architecture able to process graphs later detailed in Section 2.2 – as a key element towards achieving this goal of self-driving network. GNNs have been shown to generalize to a large variety of domains, including chemistry, physics, electronics, and networking. Graphs offer an efficient data structure for representing elements in a network, their relationships and interactions, and their configuration. Our approach – later explained in Section 2.4 – is based on tailored graphs transformations and modeling in combination with GNNs. We apply our approach to different challenges in networking.

First, we use GNNs to address the challenge of providing and speeding up performance guarantees in networks using network calculus (NC). As illustrated in Figure 1.1, current state-of-the-art NC methods generally sacrifice computational effort (and hence time spent for computing) for improvement bounds tightness. Numerical evaluations show that even on moderately sized networks with around 1000 flows – a size comparable to some industrial networks – some analyses take multiple days to compute bounds. Such computation time is often not

acceptable, especially with the advent of technologies such as Audio-Video Bridging (AVB) or Time-Sensitive Networking (TSN), where NC has become a major tool for the industrial deployment of such networks.

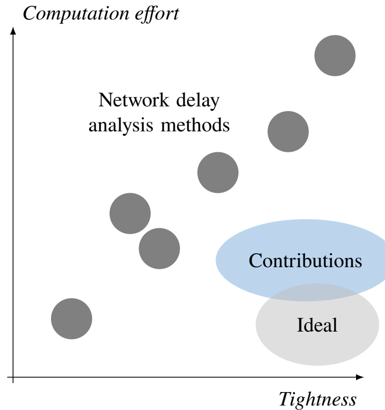


Fig. 1.1: Relationship between tightness and execution time in delay analyses

Our contributions open up a new branch of NC analyses achieving tightness close to the best state-of-the-art methods, but at a low computational cost, as illustrated in Figure 1.1. Compared to other delay analysis methods (*e.g.* schedulability analysis, model checking), most of them also suffer a similar trend where gains in tightness are usually paid with long computations. Overall, our contributions in NC will help optimize network usage and avoid over-allocation of resources when designing a network with guarantees. This is important in many industrial networks – such as avionic networks – where over-allocation of resources lead to overall inefficiencies such as too much weight and a reduction in fuel efficiency.

We also evaluate other areas where our GNN-based approach can be used. A promising area is network protocol design, where intensive engineering time is required in order to achieve specific behavior from distributed algorithms and protocols. We contribute a novel application of GNNs for the design of routing protocol, a famous challenge for distributed decisions in a network. In the scope of self-driving networks, our contribution is a first step towards machine-designed network protocols, avoiding most of the manual work required in protocol design.

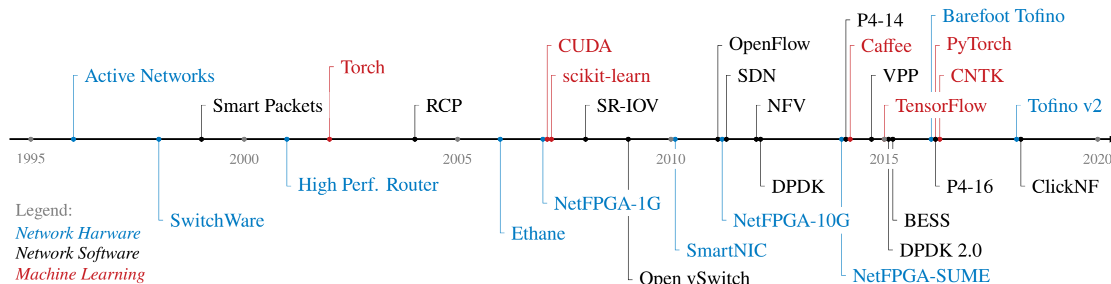


Fig. 1.2: Evolution of tools, software and hardware acceleration related to networking and machine learning in the last 25 years

In parallel to this concept of self-driving network, various advances have also been made on

processing speed in both the networking and machine learning fields in the last two decades, as illustrated in Figure 1.2. Combining advanced algorithms with efficient hardware acceleration is nowadays possible, providing advanced features for packet processing.

As illustrated earlier in Figures 1.2 and 1.3, various advances were made in the area of fast packet processing, bringing effective solutions with ease-of-use and programming flexibility. With our works, we explore how this trend relates to real-world problems such as industrial networks with needs for specific protocols and performance. We show that these new platforms can implement advanced security features at line-rate, and also execute ML inference to optimize Central Processing Unit (CPU) usage.

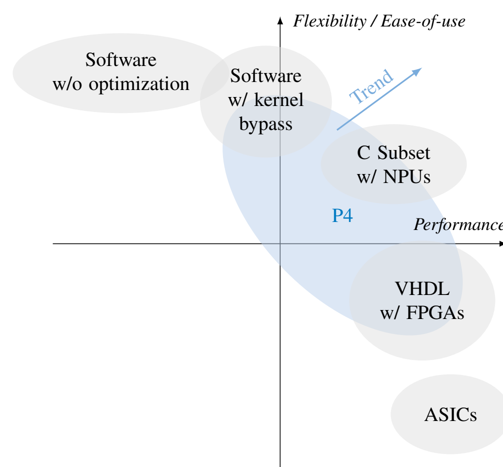


Fig. 1.3: Relationship between performance and ease-of-use of different dataplane technologies currently available

Overall, this thesis contributes an efficient graph-based paradigm for reasoning about networking challenges. We show that this paradigm is versatile and applicable in various domains, from network verification, to performance evaluation, and network protocol design. We also illustrate how advanced algorithms can be implemented in the dataplane, enabling sophisticated features to be run in networks, an essential step towards self-driven networks.

1.2 Research objectives and expected benefits

The general goal of this work is to develop the model paradigms and technological bricks necessary for achieving self-driving networks, with a focus on networks designed with the goal of performance optimization and availability.

This work is structured around the two following research objectives:

O1: Develop more generic and more efficient performance evaluation models

Early research on performance evaluation had the goal of creating clean, closed-form mathematical models of individual protocols, applications, and systems. While those models produced accurate results in various areas such as prediction of Transmission Control Protocol

(TCP) bandwidth [140], closed-form analysis can quickly become too complicated once the different parameters and configuration options of the protocol have to taken into account.

Due to the multiple-feedback loops present at different layers of the protocol stack, another challenge is interoperability between mathematical models. An example would be the challenge of taking wireless transmission channel models into account when studying higher layer protocols. Another challenge to face is to keep up with the current pace of new protocols proposals, their different implementations and different configuration parameters.

A data-driven approach might be a solution to those challenges, with the use of repeatable measurements, in conjunction with learning algorithms and representations of communication networks which are generic enough to accommodate a wide range of network protocols and architectures.

O2: Include lessons learned from data in network architectures and protocols

Current network protocols are mainly based on hard-coded rules which were optimized for specific use-cases in mind. An example of such practice is congestion control, where a common approach has been to design algorithms which are optimized to certain network conditions (eg. data-centers with low latency, satellite communication with high latency). Recent contributions such as the work from Dong *et al.* with Performance-oriented Congestion Control (PCC) in [50], or the work from Winstein and Balakrishnan with RemyCC in [189], have shown more goal and data-oriented approaches to congestion control.

Based on foundations from the previous research objective, the models developed should serve as a basis for designing network architectures, managing networks, and guiding network protocol designs.

The overall approach taken for the Habilitation work is illustrated in Figure 1.4.

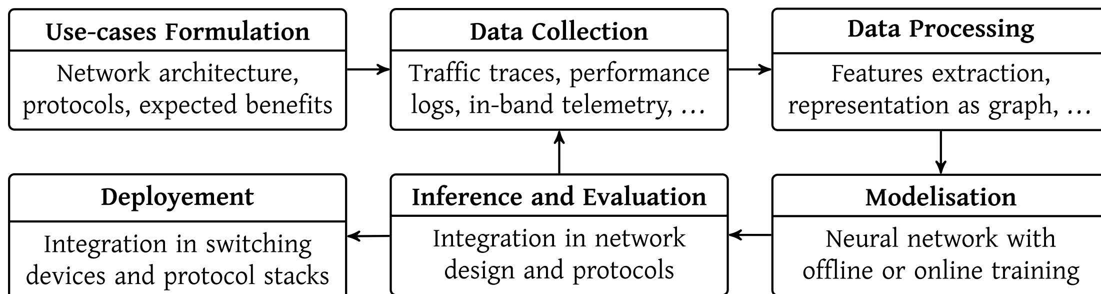


Fig. 1.4: Overview of the approach taken for the Habilitation work

1.3 Contributions and document structure

The rest of this thesis is structured into five chapter: Background, Tight and efficient bounds in network calculus with fast heuristics, Application of graph-based deep learning methods for computer networks, Hardware and software for efficient packet processing, and Conclusion.

Chapter 2 provides an overview over the three main aspects addressed in this thesis, namely NC in Section 2.1, GNNs in Section 2.2 and P4 in Section 2.3. In this chapter, we also detail the

key contribution of this thesis, namely the application of graph models to networking challenges in Section 2.4. This key contribution is a generalization and unification of the concepts contributed in our works: [60, 68, 69, 61, 62, 70, 64, 65, 75]

In **Chapter 3**, we detail our contributions in the domain of NC, where extensions of NC are proposed, as well as the first applications of GNNs to NC. Our contributions are:

- In [19, 20] and Section 3.3.1 we extend NC's network analyses to the analysis of multicast flows. We show that using a naïve approach to multicast flows leads to loose delay bound. Our contribution enables the application of well-known principles from NC such as Pay Burst Only Once (PBOO) and Pay Multiplexing Only Once (PMOO) to multicast flows.

- In [21] and Section 3.3.2, we propose flow detouring in NC, an extension of the work from Bondorf [18] which adds pessimism in the network model in order to gain tightness. Our contribution stems from the realization that the PMOO property is mostly valuable on networks where flows share a large number of servers. By virtually adding servers on the cross-flows' path, *i.e.* adding pessimism in the network, we show that delay bounds can be reduced.

- In [69] and Section 3.4 we contribute the first work applying GNNs to NC. We show that GNNs are an efficient heuristic of NC, which is used for predicting the delay bounds from various NC analyzes. Since those predictions are not formally valid, we also illustrate how they are still valuable at deciding which NC analysis will produce the tightest bound given a network.

- In [62, 64, 65] and Section 3.4.2 we built on the success from [69] and propose DeepTMA. DeepTMA is a novel approach integrating predictions from a GNN more tightly in the Tandem Matching Analysis (TMA) from Bondorf *et al.* [24], a state-of-the-art NC analysis achieve good tightness. We illustrate that TMA's exhaustive search for the best tandem decomposition can be avoided by using a GNN for predicting the most promising decomposition. Via numerical evaluation, we show that DeepTMA is able to achieve tightness close the one from TMA but with a computational effort orders of magnitude smaller.

- In [75] and Section 3.4.3 we build on the success and findings of DeepTMA and propose a DeepFP, another application of GNNs to NC. DeepFP is an approach for scaling Flow Prolongation (FP)'s property of NC from [18], which also suffers from an computationally expensive exhaustive search. DeepFP predicts the best prolongations to use, short-cutting the exhaustive search. As with DeepTMA, we show via numerical evaluation that DeepFP is able achieve good tightness at a small computational effort compared to FP.

In **Chapter 4**, we explore application of GNNs to networking challenges outside of NC. Our contributions are:

- In [60, 61] and Section 4.3 we contribute one of the first applications of GNNs to networking challenges. Our proposal, called DeepComNet, applies GNNs to the challenge of performance evaluation of flows. We show that the bandwidth of elastic flows and the end-to-end latency of flows with constant rate can be predicted. DeepComNet can be used as a what-if analysis tool for fast network design and evaluation.

- In [68] and Section 4.4 we contribute an application of GNNs to the task of network protocol generation. We show that one of key concepts behind GNNs – message passing – can be directly mapped to packets and routers, effectively using computer networks as a distributed GNN. We illustrate that this concept can be applied to packet routing.
- In [70] and Section 4.5 we contribute DeepMPLS, an application of GNNs for the verification and synthesis of Multiprotocol Label Switching (MPLS) configurations. We show that MPLS configuration can be mapped to graphs, which are then used for predicting if the MPLS configuration satisfies some properties such as not containing any loops.

In **Chapter 5**, we explore how advanced functionalities may be brought to the dataplane via P4 or software-based solutions with kernel bypass. Our contributions are:

- In [73, 71, 155] and Section 5.3 we evaluate P4 in an industrial environment both from a functional perspective and a performance perspective. We show that P4 is almost sufficient enough to implement Avionics Full DupleX Switched Ethernet (AFDX), an Ethernet-based network protocol specific to avionics. In our performance evaluation, we show that commodity hardware can be used and is sufficient for processing 10 Gbit/s traffic.
- In [157] and Section 5.4 we explore the use of hash functions if P4 on various hardware platforms: CPU, Network Processing Unit (NPU) and Field Programmable Gate Array (FPGA). We illustrate on these three different platform how hash functions can be efficient implemented and do a performance evaluation. Such extension of P4 is relevant for security-oriented applications, or simply for constructing more advanced data structures than the ones already provided by P4.
- In [139] and Section 5.5 we contribute an approach for using ML inference in the dataplane for more efficient packet processing in software routers. We illustrate that by having a dynamic batching allocation in Vector Packet Processing (VPP), CPU time can be saved. This illustrates that ML inference is possible in the dataplane, a step towards more intelligent networks.

Finally, **Chapter 6** concludes this thesis by summarizing the contributions and discussing future work and open research directions.

All the articles presented in this work can be found in Appendix A. Besides the technical contributions presented here, open datasets and tools were also contributed and listed in Appendix B. Apart from the content of this manuscript, other research works related to network design, network operation, or network performance were also performed. The full list of works can be found in Appendix A.4.

1.4 Remarks about this document

The various abbreviations and notations used throughout this thesis are listed in Appendix C. The digital version of this document also contains clickable links which are marked in gray (e.g. `https://example.com`). Some notes regarding the origin of the content of this thesis appear in the text in the following form:

Note This is a note

2. BACKGROUND

2.1 **Deterministic network calculus** 7
2.2 **Graph neural networks** 10
2.3 **P4: An approach for efficient packet processing devices** 12
2.4 **Data Driven Methods for Networking** 13

We introduce in this chapter key technical concepts which will be used through this Habilitation thesis. Additional background and related work will also be provided in the subsequent chapters.

2.1 Deterministic network calculus

Deterministic network calculus (DNC) – or often simply called network calculus (NC) – is a mathematical framework for analyzing performance guarantees of traffic flows in queuing networks. This formalism was initially developed in the early 1990's by Cruz [43, 44], with the so-called (σ, ρ)-calculus. Its current predominant application is computer networks. We will describe here some of the main mathematical results of this framework, and make a parallel with its current application in Ethernet networks. For a more thorough description of NC, we refer to the books from Chang [40], from Le Boudec and Thiran [114], or from Bouillard *et al.* [30].

In this framework, flows are described as their cumulative arrival of data per unit of time. This is modeled by a non-decreasing function of time (noted t) into the set of monotonically-increasing and strictly positive functions $\mathcal{F}$, more formally defined as:

$$\mathcal{F} = \{f : \mathbb{R}^+ \rightarrow \mathbb{R}^+ | \; \forall 0 \leq t \leq s : f(t) \leq f(s), f(0) = 0\} \tag{2.1}$$

This function is called its *cumulative arrival function*. A flow with cumulative arrival function R, or more simply a flow R, is said to be (σ, ρ)-upper constrained if ([40, Definition 1.1.1]):

$$R(t) - R(s) \leq \rho \cdot (t - s) + \sigma, \text{ for all } 0 \leq s \leq t, \text{with } \rho, \sigma \text{ constant values.} \tag{2.2}$$

In Equation (2.2), σ is called the burstiness parameter, and ρ the upper bound on the long-term average rate of the traffic flow. This curve is illustrated in Figure 2.1 with the gray curve. In a real network, a flow can be forced to follow a prescribed (σ, ρ) constraint using a so-called *shaper*.

More generally, a flow R is said to have a deterministic arrival curve $\alpha \in \mathcal{F}$ if its cumulative arrival function R satisfies for all s and t such that for all $0 \leq s \leq t$ ([114, Definition 1.2.1]):

$$R(t) - R(s) \leq \alpha(t - s) \tag{2.3}$$

In non-mathematical words, it defines the worst-case behavior of a flow by a well-known function.

Network elements representing queues or switches, often called *servers* in NC terms, impose a service curve $\beta \in \mathcal{F}$ on an input flow R, such that the output flow R^* is defined by ([114, Definition 1.3.1]):

$$R^*(t) \geq \inf_{0 \leq s \leq t} \{R(t-s) + \beta(s)\} \tag{2.4}$$

The operation on the right hand-side of the inequality is known as the min-plus convolution, and is part of the min-plus algebra used in NC. The min-plus convolution is noted as $\otimes$, such that $R^*(t) \geq (R \otimes \beta)(t)$ ([114, Definition 3.1.10]). The second operation of the min-plus algebra is the deconvolution, noted $\oslash$, which is defined as ([114, Definition 3.1.13]):

$$(f \oslash g)(t) = \sup_{0 \leq s} \{f(t+s) - g(s)\} \tag{2.5}$$

In case of an Ethernet interface with link speed C and delay δ, the service curve β can be expressed as:

$$\beta(t) = C[t-\delta]^+, \text{where } [x]^+ = \max(0, x) \tag{2.6}$$

This particular affine curve is called a rate-latency service curve, and it is illustrated in Figure 2.1 with the black curve.

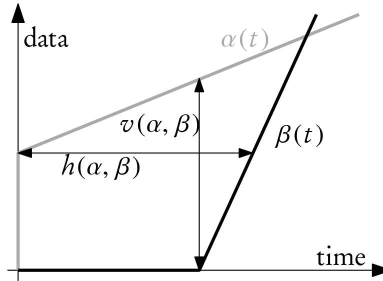


Fig. 2.1: Latency ($h(\alpha, \beta)$) and buffer ($v(\alpha, \beta)$) bounds in DNC

Using this formalism, two performance bounds can be derived, as presented in Figure 2.1:

The backlog bound or queue size bound, which corresponds to the maximal vertical deviation between the arrival and service curve $v(\alpha, \beta)$, or in mathematical terms ([114, Theorem 1.4.1]):

$$R(t) - R^*(t) \leq \sup_{s \geq 0} \{\alpha(s) - \beta(s)\} = (\alpha \oslash \beta)(0) = v(\alpha, \beta) \tag{2.7}$$

The delay bound which corresponds to the maximal horizontal deviation between the arrival and service curves $h(\alpha, \beta)$, or in mathematical terms ([114, Theorem 1.4.2]):

$$R^*(t) - R^*(t-s) \leq \sup_{t \geq 0} \left\{ \inf_{s \geq 0} \{\alpha(t) \leq \beta(t+s)\} \right\} = h(\alpha, \beta) \tag{2.8}$$

One strong property of NC is the so-called *concatenation*, where a large network of servers can be simplified to a single server using the min-plus convolution. A simple example is a flow

traversing two servers with respective service curves β_1 and β_2. It can be shown (see [114, Theorem 1.4.6]) that this is equivalent to a flow traversing a single server with service curve $\beta_C = \beta_1 \otimes \beta_2$, and the resulting bounds will be tighter than an analysis done with β_1 and β_2 separately.

Given a strict service curve that guarantees a minimum output of β if data is present at a server, we can also lower bound the service left-over for a specific flow:

Theorem 2.1 (Left-over service curve)**.** *Consider a server s that offers a strict service curve β. Let s be crossed by flows f_0 and f_1, with arrival curves α_0, respectively α_1. Then the worst-case residual resource share under arbitrary multiplexing of f_1 at s is:*

$$\beta^{\text{l.o.}f_1} = \beta \ominus \alpha_0$$

with $(\beta \ominus \alpha)(d) = \sup\{(\beta - \alpha)(u) \mid 0 \leq u \leq d\}$ denoting the non-decreasing upper closure of $(\beta - \alpha)(d)$.

Regarding packet scheduling, NC can be used on networks with various algorithms such as priority-based scheduler (see the book from Le Boudec and Thiran [114, Chapters 2.4, 6 and 7]) or fair queuing (see the work from Stiliadis and Varma in [166]).

One of the pitfalls of NC is its looseness, which is generally attributed to a loose handling of flow multiplexing. Various methods have been proposed to address this issue, such as the *Pay Burst Only Once (PBOO)* (see book from Le Boudec and Thiran [114]) and *Pay Multiplexing Only Once (PMOO)* (see work from Schmitt *et al.* [153]).

We present here some details on the most notables analysis from the literature using those properties. Using the definitions and theorems presented above, the end-to-end performances of flows interacting on a network of servers can be computed. We call the analyzed flow *flow of interest (foi)*.

Total Flow Analysis (TFA) [114]

The TFA first computes per-server delay bounds. Each one holds for the sum of all the traffic arriving to a server, i.e., these bounds are independent of the foi. The flow's end-to-end delay bound is derived by summing up the individual server delay bounds on its path. The TFA's server-isolating approach constitutes a direct application of Equations (2.7) and (2.8); it is known to be inferior to the following analyses [114, 154].

Separate Flow Analysis (SFA) [114]

The SFA is a direct application of other theorems: first compute the left-over service of each server on the foi's path using Theorem 2.1, then concatenate them and finally derive the end-to-end delay bound using Equations (2.7) and (2.8). Deriving the end-to-end delay bound using only one service curve will consider the burst term of the foi only once, using the PBOO property.

Pay Multiplexing Only Once (PMOO) [154]

The PMOO analysis first convolves the tandem of servers before subtracting the cross-traffic. Using this order, the bursts of the cross-traffic appear only a single time compared to the SFA analysis where the bursts are included at each server. Therefore, multiplexing with cross-traffic is only paid for once. However, [153] showed that the PMOO method does not necessarily outperform the SFA.

A numerical analysis and comparison of those different methods is performed later in Chapter 3. Additional network analyses from the state of the art are also reviewed in Chapter 3.

We note that due to the way flows are modeled, elastic protocols such as Transmission Control Protocol (TCP) are hard in practice to use with this framework. Various propositions were made regarding using NC on feedback-based protocols such as TCP, such as for the instance the works from Chang [39], from Baccelli and Hong [9], or from Agrawal *et al.* [3]. We note that those previous work are either limited to the study of a single flow, or use an idealized version of TCP.

Details about how NC can be applied to Ethernet networks and how it has been used during the development of the A380 in the early 2000's are presented by Grieu [79]. This work has led to the definition of the Avionics Full DupleX Switched Ethernet (AFDX) standard [2].

Various tools are available for the performance evaluation of network with NC, open-source ones such as the NCorg Network Calculator from Bondorf and Schmitt [22], the DISCO Network Calculator from Schmitt and Zdarsky [152], CyNC from Schioler *et al.* [150], COINC from Bouillard *et al.* [28], or NC-maude from Boyer [32]; or closed-source and commercial ones targeted at the industry (*e.g.* [131, 35, 103]). A recent survey of NC tools and their application to real-time systems was recently published by Zhou *et al.* [200].

2.2 Graph neural networks

In this section, we detail the neural network (NN) architecture used for training NNs on graphs, namely the family of architectures initially proposed by Gori *et al.* [77], Scarselli *et al.* [148] and based on graph neural networks (GNNs).

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $(v, u) \in \mathcal{E}$. Let $\mathbf{i}_v \in \mathbb{R}^n$ and $\mathbf{o}_v \in R^m$ represent respectively the input features (e.g. node type, service or arrival curve parameters) and output values for node v (e.g. decision for the NC analysis). The concept behind GNNs is called *message passing*, where so-called hidden representations of nodes $\mathbf{h}_v \in \mathbb{R}^k$ are iteratively passed between neighboring nodes. Those hidden representations are propagated throughout the graph using multiple iterations until a fixed point is found or after a fixed number of iterations. The final hidden representation is then used for predicting properties about nodes. This concept can be formalized as:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_u^{(t-1)} \;\middle|\; u \in \textsc{Nbr}(v)\right\}\right) \tag{2.9}$$

$$\mathbf{o}_v = out\left(\mathbf{h}_v^{(t\to\infty)}\right) \tag{2.10}$$

$$\mathbf{h}_v^{(t=0)} = init\left(\mathbf{i}_v\right) \tag{2.11}$$

with $\mathbf{h}_v^{(t)}$ representing the hidden representation of node v at iteration t, $aggr$ a function which aggregates the set of hidden representations of the neighboring nodes $\textsc{Nbr}(v)$ of v, out a function transforming the final hidden representation to the target values, and $init$ a function for initializing the hidden representations based on the input features.

The concrete implementations of the $aggr$ and out functions are feed-forward neural net-

works (FFNNs), with the addition that $aggr$ is the sum of per-edge terms [148], such that:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_{\text{NBR}(v)}^{(t-1)}\right\}\right) = f\left(\sum_{u \in \text{NBR}(v)} \mathbf{h}_u^{(t-1)}\right) \tag{2.12}$$

with f a FFNN. For $init$, a one-layer FFNN is used to fit the input features to the dimensions of the hidden representations.

Gated graph neural networks (GGNNs) were recently proposed by Li *et al.* [119] as an extension of GNNs to improve their training. This extension implements f using the Gated Recurrent Unit (GRU) memory unit from Cho *et al.* [42] and unrolls Equation (2.9) for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing. The neural network architecture is illustrated in Figure 2.2.

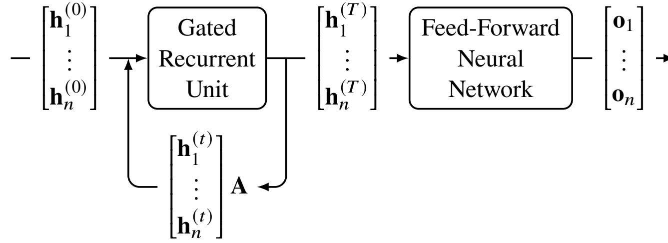


Fig. 2.2: Overview of GGNN architecture

Formally, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \tag{2.13}$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z \mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \tag{2.14}$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r \mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \tag{2.15}$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W} x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \tag{2.16}$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \tag{2.17}$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}$ and $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}$ are trainable weight matrices, and $\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is the graph adjacency matrix, determining the edges in the graph $\mathcal{G}$.

Equation (2.13) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for GNNs in Equations (2.9) and (2.12). Equations (2.14) to (2.17) correspond to the mathematical formulation of a GRU cell [42], with Equation (2.14) representing the GRU reset gate vector, Equation (2.15) the GRU update gate vector, and Equation (2.17) the GRU output. In order to propagate the hidden representations throughout the complete graph, a fixed number of iterations of Equations (2.14) to (2.17) are performed. This extension has been shown to outperform standard GNN which require to run the iteration until a fixed point is found.

Additionally, this NN architecture can be extended with an attention mechanism similar to the one proposed by Veličković *et al.* [182]. Thus, the GNN can give preference to some neighbors over other ones via a trained function. For each edge (v, u) in the graph, we define a weight parameter $\rho_{v,u}^{(t)}$ depending on the concatenation of $\mathbf{h}_v^{(t)}$ and $\mathbf{h}_u^{(t)}$:

$$\rho_{v,u}^{(t)} = \sigma\left(\mathbf{W}_a \left\{\mathbf{h}_v^{(t)}, \mathbf{h}_u^{(t)}\right\} + \mathbf{b}_a\right) \tag{2.18}$$

with trainable weights $\mathbf{W}_a$ and bias parameters $\mathbf{b}_a$. Equation (2.12) can then be rewritten as

$$\mathbf{h}_v^{(t)} = \sum_{u \in \textsc{Nbr}(v)} \rho_{v,u}^{(t-1)} f\left(\mathbf{h}_u^{(t-1)}\right). \tag{2.19}$$

GNNs attracted lots of work in the machine learning (ML) community. For a good overview over some recent improvements and an attempt at a general unification of those improvements, we refer to the work from Battaglia *et al.* [12]. Various implementations were also open-sources, such as the works from Fey and Lenssen [55], from Wang *et al.* [185], and from Grattarola and Alippi [78].

2.3 P4: An approach for efficient packet processing devices

In conjunction with the current trend towards softwarization of functionalities in the field of communication networks with the advent of Software Defined Networking (SDN) and related technologies, a recent development from Bosshart *et al.* [27] called P4 – Programming Protocol-Independent Packet Processor – proposes a flexible way to specify packet processing devices. A full review on dataplane programmability is provided later in Section 5.2.

The main promises of the P4 programming language and toolchain are:

1. A simple specification of packet processing pipelines using a high-level Domain Specific Language (DSL), requiring no expert knowledge about the final hardware. This DSL was specially designed to be expressive enough for the various actions necessary in network protocols, while restrictive enough to enable simple compilation to various dedicated target hardware. Sample snippets of P4 descriptions for standard Ethernet and IPv4 routing are given in Listings 2.1 and 2.2. The complete specification of the P4 language is available on the P4 website [170, 171] and also reviewed by Budiu and Dodd [38].

2. Compilation of specification for different hardware targets, ranging from hardware solutions such as Field Programmable Gate Arrays (FPGAs) or Network Processing Units (NPUs), to finally purely software solutions targeting multi-core and many-core processors;

3. Reconfigurability in order to modify the behavior of packet-processing devices in the field;

4. Possibility to test packet processing pipelines using well-known network emulation tools such as mininet [112] and ability to emulate complete network architectures.

This approach is also in line with model driven engineering, where high level descriptions of systems are used in order to formally verify various properties of systems.

Currently, two different P4 standards are evolving in parallel: $P4_{14}$, which is the original P4 and subject of this paper, and $P4_{16}$, a major redesign of the language with an object oriented approach.

Listing 2.1: Example of Ethernet frame format definition in P4

```
header_type ethernet_t {
  fields {
    dstAddr   : 48;
    srcAddr   : 48;
    etherType : 16;
  }
}
```

Listing 2.2: Example of IPv4 packet routing in P4

```
action route_ipv4(dst_port, dst_mac, src_mac, vid) {
  modify_field(standard_metadata.egress_spec, dst_port);
  modify_field(ethernet.dst_addr, dst_mac);
  modify_field(ethernet.src_addr, src_mac);
  modify_field(vlan_tag.vid, vid);
  add_to_field(ipv4.ttl, -1);
}
```

P4 uses a generic packet processing pipeline as a basis called *abstract forwarding model*. This model applied to a switch is illustrated here in Figure 2.3. Packets are first parsed according to customizable frame format definitions.

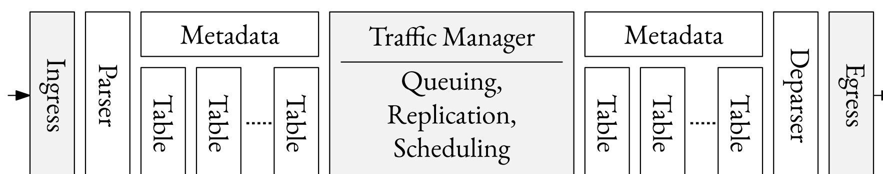


Fig. 2.3: P4 Abstract Forwarding Model of a switch

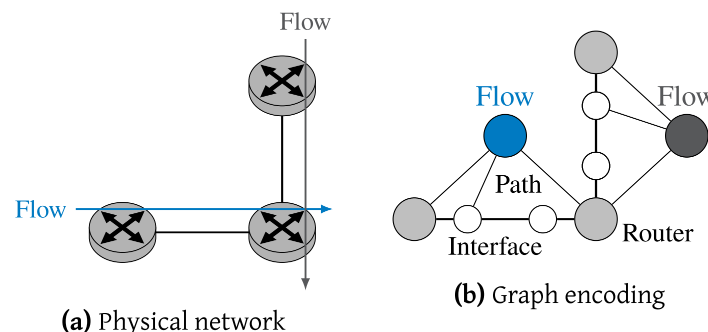
Based on the fields and associated values of the protocols, so-called *match+action tables* are used in order to process packets. Available actions include packet modification (changing field value, adding or removing headers), replication (for broadcast or multicast), dropping packets or triggering of flow control (namely update of action tables such as counters or policers). Those match+action tables are conceptually similar to the ones used in OpenFlow switches.

2.4 Data Driven Methods for Networking

We present in this section one of the key contributions of this Habilitation. With [60], we introduced one of the first GNN application to the field of networking and framework on how to model networking problems and topologies using graphs. This early work – presented later in Section 4.3 – has lead to various other publications with applications in different fields of networking as presented in Chapters 3 and 4.

When applying ML, one of the keys to successful and accurate predictions is to have a tailored data-structure for inputs representing the problem being solved, as-well-as a ML algorithm able to make use of this structure. In image processing for example, a stepping stone contribution from Le Cun *et al.* [115, 116] was to apply convolutional neural networks (CNNs), a NN architecture able to process images as tiles, *i.e.* taking advantage of pixels and their neighbors in a hierarchical way in 2-dimensional matrices.

In the case of computer networks, there is a need for an efficient data-structure able to accurately represent topologies, flows and their properties. While there has been attempts at using traditional FFNN and CNN at networking problems (with some successes), those architectures and data-structures are not tailored for representing data links and their relationships. We propose to use (undirected) graphs and GNNs for this task, since they are a natural data-structure and NN architecture for representing and processing network topologies. Adding and processing information such as flows can be easily performed using dedicated nodes with edges connecting the flows to their path, as illustrated in Figure 2.4.



(a) Physical network

(b) Graph encoding

Fig. 2.4: Overview of graph transformation process

Overall, our process for solving networking tasks using GNNs is split into different steps:

Graph tranformation In this first step, the information required for solving the network problem is represented as a graph. This includes representing relevant elements from a network topology such as switches or routers, their internal queues, network flows, and their configuration. As shown later in Section 4.5 with Multiprotocol Label Switching (MPLS), even advanced network protocol configuration can be included in the graph. This task usually requires expert knowledge about the problem in order to build a relevant graph and correctly link knowledge in the graph.

GNN training Once the graph transformation is properly defined, GNN training on the graphs is performed. In this step, the networking problem is mapped onto a ML task on the graph, namely classification or regression on specific nodes or edges in the graph. For example, in case of performance evaluation of flows (as done in [60] and Section 4.3) where numerical performance indicators of flows need to be produced, regression is performed on flow nodes. ML expertise is required for this step, since a GNN needs to be trained, either following a supervised approach as done in all works presented in Chapters 3 and 4, or alternate methods (*e.g.* reinforcement learning). This requires defining proper datasets and training algorithms.

Inference and action In this last step, inference is performed using the trained GNN and the

predictions are included back in the networking problem. We illustrate for example in Section 3.4 how predictions are fed back to formal methods, or in Section 4.4 how it can be used to generate network protocols. Inclusion into a SDN controller can also be envisioned, where GNNs may be used for dynamically controlling certain aspects such as Traffic Engineering (TE) in a network.

We will illustrate in Chapters 3 and 4 different applications of this data-driven approach for networking, from formal analysis of networks to design of network protocols.

3. TIGHT AND EFFICIENT BOUNDS IN NETWORK CALCULUS WITH FAST HEURISTICS

3.1 Introduction 17
3.2 Related work 18
3.2.1 Flow Prolongation 18
3.2.2 Tandem Matching Analysis 19
3.2.3 NC Combined with other methodologies 20
3.2.4 Related approaches 20
3.3 Extending network calculus analyses for tighter bounds 20
3.3.1 Multicast flow analyses in DNC 21
3.3.2 Flow detouring 23
3.4 Graph-based deep learning for speeding up formal verification 25
3.4.1 A heuristic for different NC analyses 25
3.4.2 Predicting best tandem decompositions with DeepTMA 29
3.4.3 Predicting best flow prolongations with DeepFP 32
3.5 Conclusion on network calculus improvements 36

3.1 Introduction

We introduced deterministic network calculus (DNC) in Section 2.1 as a formal method of choice for validating networks. In the last few years, this method has gained lots of attention in the scope of Ethernet networks, first with Avionics Full DupleX Switched Ethernet (AFDX) in the early 2000 (*e.g.* works from Grieu [79], Frances *et al.* [57], Boyer and Fraboul [33], Adnan *et al.* [1]), and now with Audio-Video Bridging (AVB) and Time-Sensitive Networking (TSN), with various works modeling AVB's and TSN's flows and schedulers (*e.g.* works from Queck [143], De Azua and Boyer [48], Le Boudec [113], Daigmorte *et al.* [45], Zhao *et al.* [198, 199], Mohammadpour *et al.* [135], Maile *et al.* [124]).

DNC as a research topic can be split into various sub-topics. The first sub-topic is concerned with correctly modeling flows and schedulers according to their specifications, namely deriving correct arrival and service curves. The second sub-topic is more concerned with the analysis of the server graph, namely improving the methods such as Total Flow Analysis (TFA), Separate Flow Analysis (SFA) or Pay Multiplexing Only Once (PMOO), which are (for the most part) agnostic to the characteristics of the arrival or service curves. In both cases, most of research is done on getting the tightest bound possible, *i.e.* the closest bound to the worst-case delay.

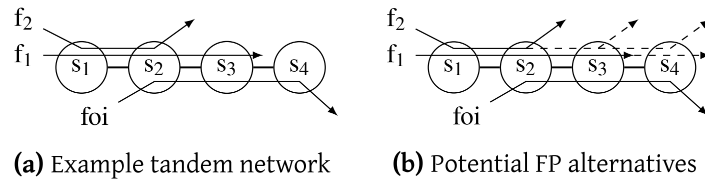
In the scope of this chapter, we focus on the second sub-topic. Gains in the analysis side of DNC can be directly beneficial to a large variety of use-cases since they are agnostic to the type of arrival or service curves, while dedicated works on arrival or service curves have a more limited scope. We look into DNC's analyses and explore how to extend them to multicast flows in Section 3.3.1, make them tighter with a counter-intuitive property in Section 3.3.2, or make them faster with minimal loss on tightness with deep learning in Section 3.4.

3.2 Related work

3.2.1 Flow Prolongation

An approach to overcome one of PMOO's looseness in some networks was recently presented by Bondorf [18] with the Flow Prolongation (FP) feature. It actively converts the network model given to the network calculus (NC) analysis to a more pessimistic one that circumvents limitations of the NC analysis capabilities.

FP is conceptually straight-forward: assume cross-flows take more hops than they actually do. Take the sample tandem in Figure 3.1a, where bounding the arrivals of data flows f_1 and f_2 is required at their first location of interference with the flow of interest (foi), server s_2. Assuming arbitrary multiplexing, Bondorf and Schmitt [23] showed that the PMOO analysis suffers from the segregation effect, both flows assume to only receive service after the respective other flow was forwarded by server s_1 – an unattainable pessimistic forwarding scenario in the analysis-internal view on the network. FP tries to steer the analysis such that it does not have to apply this pessimism by prolonging flows inside the analysis: the dashed lines in Figure 3.1b depict potential prolongations of the two flows' paths. Each prolongation alternative that matches their sinks will allow for their aggregate treatment at s_1, mitigating the problem. Yet, this adds interference to the foi.



(a) Example tandem network **(b)** Potential FP alternatives

Fig. 3.1: Illustration of the FP feature of NC

Therefore, we search for the best prolongation alternative trading off both aspects. Unfortunately, finding the best prolongation alternative is prone to a combinatorial explosion. On each tandem of length n with m cross-flows, there are $O(n^m)$ alternatives to prolong flows. Even with a deep understanding of the NC analysis applied to reduce FP alternatives it could not be made to scale to larger models [18]. Nonetheless FP is a powerful feature to add to a NC analysis, it was even adopted in the stochastic network calculus (SNC) by Nikolaus and Schmitt [138].

We will show in Section 3.4.3 that the exhaustive search of FP can be avoided, leading to tight bounds at low computational effort.

3.2.2 Tandem Matching Analysis

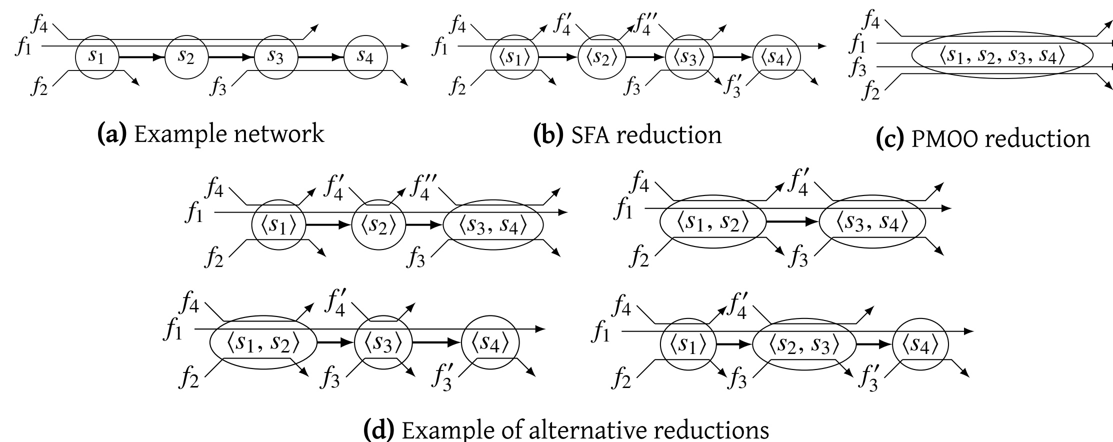
Another approach to overcome the looseness of SFA and PMOO is the so-called Tandem Matching Analysis (TMA) which was recently proposed by Bondorf *et al.* [24]. The central idea behind TMA is that NC's concatenation property can be interpreted as a network transformation rule, which can be selectively applied at different points in a network. The central means of transforming tandems is *cutting*:

Definition 3.1 (Cutting NC Tandems)**.** *Given a tandem $\mathcal{T} = (\mathcal{V}_{NC}, \mathcal{E}_{NC}, \mathcal{F})$ and a NC analysis $\mathcal{A}$, a cut marks edge $e \in \mathcal{E}_{NC}$ such that $\mathcal{A}$ will analyze $\mathcal{T}$ as a sequence of sub-tandems $\langle \mathcal{T}_l, \mathcal{T}_r \rangle$ where $\mathcal{T}_l$ holds all the model information to the left of e and $\mathcal{T}_r$ that to the right of e. A cutting (also called combination of cuts) is a set of cuts on $\mathcal{T}$.*

Visually, the analyses implementing Pay Burst Only Once (PBOO) and PMOO proceed as depicted in Figure 3.2b and 3.2c:

All Cuts (Figure 3.2b): SFA cuts every edge in the NC model along with the flows crossing it (except the foi f_1). The resulting sub-tandems are demarcated with $\langle\cdot\rangle$ and consist of single servers. For the cut flows, their arrivals at the subsequent server need to be bounded (we denoted the respective location with f_i'). Such a flow's bound consists of its initial – given burstiness – bound plus the worst-case increase due to having crossed the previous servers.

No Cuts (Figure 3.2c): Without cuts, the entire tandem is analyzed at once. Mitigating the need for deriving bounds on flow arrivals in the network allows for achieving the PMOO property in addition to PBOO.



(a) Example network **(b)** SFA reduction **(c)** PMOO reduction

(d) Example of alternative reductions

Fig. 3.2: Illustration of reduction rules applied by TMA when analyzing f_1

TMA extends these two rules and explores alternative reductions, as illustrated in Figure 3.2d. In concrete terms, TMA performs an exhaustive search over all combinations of the reduction rules above, an keeps the combination leading to the tightest delay bound. Thus, the best reduction rules for a given flow can be found.

While TMA provides good tightness, this comes at a computational cost. The amount of alternatives on a single tandem of n servers is 2^{n-1}. TMA provides a recursive algorithm whose execution time can exceed several hours, e.g., when analyzing networks with >1000 servers and four times as many flows [24].

We will show in Section 3.4.2 that the exhaustive search of TMA can be avoided, leading to tight bounds at low computational effort.

3.2.3 NC Combined with other methodologies

The (min,+)-algebraic NC provides deterministic modeling and analysis techniques. It has seen various efforts to extend NC's capabilities. For instance, the underlying (min,+) algebra can be exchanged for (min,×) for fading channel analysis as shown by Al-Zubaidy *et al.* [4], or for (max,+) to better fit discrete event systems as shown by Liebeherr [120]. Moreover, Boyer and Roux [34] developed a common model for NC and event stream theory has been, and Lampka *et al.* [107] showed that state-based system modeling can be integrated by pairing NC with timed automata.

NC has been used by Thiele *et al.* [174] to describe component models commonly found in real-time systems. Delay bounds can then be derived from a combination of component characteristics and the network calculus model. For example, knowledge about the busy period of a greedy processing component has been used to speed up NC computations by Guan and Yi [80] and Lampka *et al.* [108, 109].

An optimization formulation called Unique Linear Program (ULP) was proposed by Bouillard *et al.* [29], which formulate the NC model as a series for linear programs (LPs) which computes tight bounds in networks without assumptions on the multiplexing of flows. It first derives the dependencies between busy periods of servers in order to partially order the mutual impact of flows. The tight analysis requires to expand this order to all compatible total orders. There are several algorithms to solve this challenge. As shown by Bondorf *et al.* [24], the resulting amount of total orders and therefore LPs to solve can quickly becoming prohibitive. Bouillard *et al.* [29] proposed a heuristic that skips the expansion step and still derives valid bounds. Its computational demand was numerically evaluated by Bondorf *et al.* [24].

3.2.4 Related approaches

The Trajectory Approach (TA) is an adaptation to the study of network delays of the holistic approach from Tindell and Clark [176]. It was originally developed to give bounds on the scheduling of tasks on a processor. The approach was initially proposed by Migge [132] and later extended to First In First Out (FIFO) systems by Martin and Minet [127]. Bauer *et al.* [13] applied TA to the study of avionic networks with multicast flows.

The Forward End-To-End Delay Approach (FA) has been proposed more recently by Kemayo *et al.* [101]. It addresses the shortcomings of the TA. Similarly to the TA, FA is also an adaptation of the holistic approach to the case of FIFO networks. [101] and [102] applied the FA to the performance evaluation of avionic networks with multicast flows.

3.3 Extending network calculus analyses for tighter bounds

In this section we focus on two extensions of algebraic NC for enabling tighter bounds. We first focus in Section 3.3.1 on bringing the analysis of multicast flows to NC. Then we look at a counter-intuitive network transformation in Section 3.3.2, which despite adding pessimism in

the network model, enables tighter bounds.

3.3.1 Multicast flow analyses in DNC

Note This section is based on [19], published in *Proceedings of the 10th International Conference on Performance Evaluation Methodologies and Tools*, 2016, and [20], published in *Systems Modeling: Methodologies and Tools*, 2019. The complete works are in Appendices A.1.1 and A.1.2.

An important property of industrial networks is that communications are usually based on the multicast paradigm, where packets being sent by one sender are duplicated by switching elements in the network and received by multiple receivers. Using DNC on such multicast protocols requires some adaptations, since this method is restricted to the analysis of unicast communications.

Previous attempts for using DNC to analyze multicast communications only circumvented its current restriction. They do not provide a solution to overcome this limitation. Those approaches cannot benefit from all DNC capabilities to provide accurate end-to-end guarantees and networks designed based on them will be over-dimensioned.

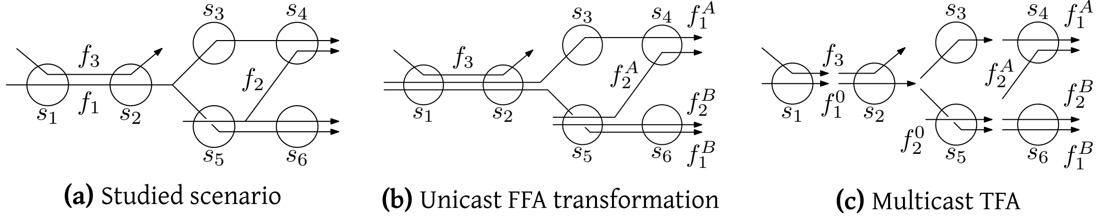
This open issue of multicast flow analysis with DNC is addressed in this section and detailed in [19, 20]. Two approaches were contributed that turn out to be steps generalizing existing analyses. The first one, Explicit Intermediate Bounds (EIB), is an approach where multicast flows are cut into sequences of unicast sub-flows. End-to-end performance bounds are then derived from sub-flow results. It does not require a transformation of the network, however, it amends a step to the analysis. Our second generalization, finally leads to a DNC multicast Feed-Forward Analysis (mcastFFA). Neither transforming the network nor cutting any flows is required. Therefore, more accurate bounds can be obtained since existing DNC principles can be applied in order to reduce effects such as flow multiplexing or burstiness. Those two approaches are illustrated in Figure 3.4, where the network with 1 unicast and 2 multicast flows depicted in Figure 3.3a is studied.

A first naïve approach to circumvent this issue is to transform a multicast flow to a set of unicast flows. Each trajectory will become one independent unicast flow, as illustrated in Figure 3.3b. The foremost problem of this approach is its overly pessimistic assumption about resource demand of multicast flows. On common sub-paths of a multicast flows' trajectories, *i.e.*, the servers before a fork, multiple unicast flows compete for resources.

In the case of TFA, Grieu [79] proposed a procedure to apply the TFA in the analysis of multicast flows, illustrated in Figure 3.3c. Flows are cut between all servers, the arrivals are aggregated and a server-local delay bound is computed. In a second step, the server delay bounds on the trajectory of interest are summed up. While this method is less pessimistic than the first naïve one, it does not benefit from NC's PBOO or PMOO properties.

Proposed approaches

The EIB analysis is the generalization of the multicast TFA analysis, illustrated in Figure 3.4a. We adapt the model such that we can analyze it with traditional NC tools, by cutting multicast flows after forking locations only; not after every server. These are the locations of explicit



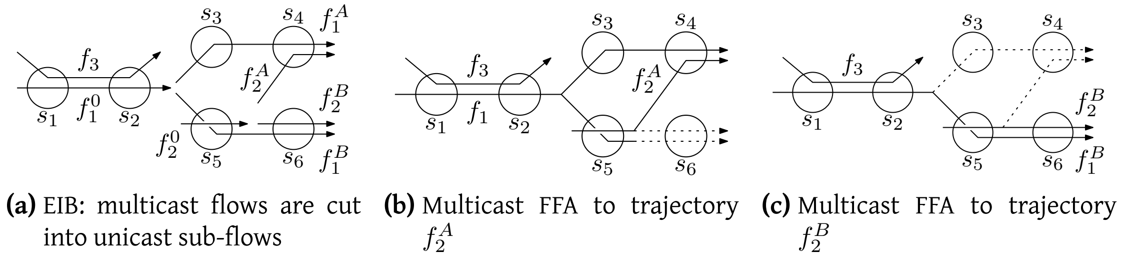
(a) Studied scenario **(b)** Unicast FFA transformation **(c)** Multicast TFA

Fig. 3.3: Example network and naïve multicast analyses

intermediate bounds The traditional SFA and PMOO analyses can then be applied on those sub-flows.

We note that the concept behind EIB of cutting a tandem in a series of sub-tandems and sub-flows shares some similarities with TMA introduced in Section 3.2.2 and Definition 3.1. While the goal of cutting differs between EIB and TMA, both methods share the same idea of network transformation rule.

For our second approach, whereas the EIB required to explicitly consider each location a multicast flow forks, the mcastFFA implicitly restricts the analysis to the trajectory relevant for the analysis. This is illustrated in Figures 3.4b and 3.4c. This step may constitute considerable effort in large networks, but is based on a backtracking of dependencies to alleviate this effort. Dependencies of a flow on others are derived by traversing the network in the opposite direction of links. The entire Feed-Forward Analysis (FFA) starts this procedure with the flow of interest. Our mcastFFA will iterate over all n trajectory of interest and execute separate analyses. Multicast cross-flows are traversed backwards, too, such that their fork locations do not enforce to cut the tandem to analyze; the relevant trajectory of the cross-flow is known and can be treated similar to a unicast cross-flow. NC's PBOO or PMOO properties can thus be used.



(a) EIB: multicast flows are cut into unicast sub-flows **(b)** Multicast FFA to trajectory f_2^A **(c)** Multicast FFA to trajectory f_2^B

Fig. 3.4: Contributed multicast analyses of flows

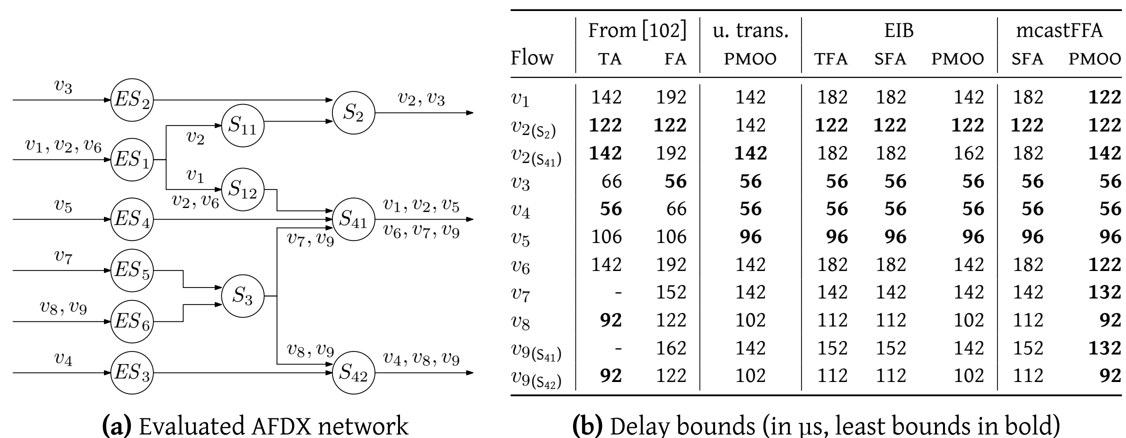
Numerical evaluation

For the numerical evaluation of the proposed approaches, we focus on the industrial network, illustrated in Figure 3.5a with two multicast flows (v_2 and v_9). Numerical results on the end-to-end delay bounds of the different flows are shown in Figure 3.5b.

Key observations w.r.t. the performance of DNC analyses confirm our theoretical evaluations. mcastFFA with PMOO produces gains of and 13.08 % compared to the multicast TFA. mcastFFA produces more accurate bounds than the EIB analysis, since it can operate on longer

tandems.

We also observe that mcastFFA results are never inferior to TA or FA introduced earlier in Section 3.2. Moreover, cases of equal results often coincide with the simplistic ones where even mutlicast TFA is competitive.



(a) Evaluated AFDX network

Flow	From [102]		u. trans.	EIB			mcastFFA	
	TA	FA	PMOO	TFA	SFA	PMOO	SFA	PMOO
v_1	142	192	142	182	182	142	182	**122**
$v_{2(S_2)}$	**122**	**122**	142	**122**	**122**	**122**	**122**	**122**
$v_{2(S_{41})}$	**142**	192	**142**	182	182	162	182	**142**
v_3	66	**56**	**56**	**56**	**56**	**56**	**56**	**56**
v_4	**56**	66	**56**	**56**	**56**	**56**	**56**	**56**
v_5	106	106	**96**	**96**	**96**	**96**	**96**	**96**
v_6	142	192	142	182	182	142	182	**122**
v_7	-	152	142	142	142	142	142	**132**
v_8	**92**	122	102	112	112	102	112	**92**
$v_{9(S_{41})}$	-	162	142	152	152	142	152	**132**
$v_{9(S_{42})}$	**92**	122	102	112	112	102	112	**92**

(b) Delay bounds (in µs, least bounds in bold)

Fig. 3.5: AFDX network evaluation of [102], extended with DNC's EIB and mcastFFA delay bounds.

Conclusion

In [19] and [20], we tackled the problem of analyzing multicast flows with DNC, a framework initially tailored to the analysis of unicast flows. Previous approaches for tried to adjust to this restriction by, *e.g.*, pessimistic re-modeling of the network, but this ultimately leads to inaccurate performance bounds.

With our two contributions, the EIB analysis and the mcastFFA, we took two crucial steps to achieve a true DNC analysis of multicast flows. In theoretical and numerical evaluations we showed in [19] and [20] that we contributed a single best DNC analysis for multicast flows. Our contribution has the flexibility to be combined with any DNC tandem analysis and improvement thereof, such as FIFO multiplexing service analysis, or packetization.

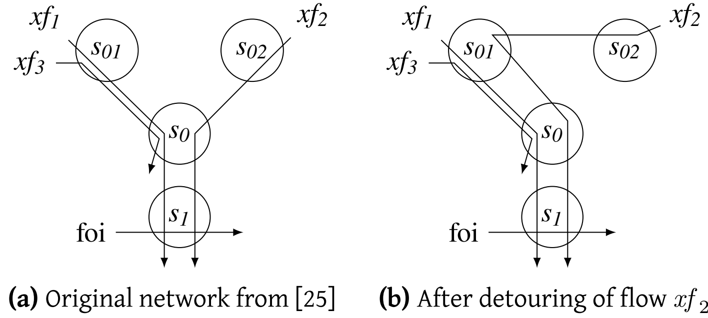
3.3.2 Flow detouring

Note This section is based on [21], published in *IFIP Networking 2020*, 2020. The complete work is in Appendix A.1.3.

We introduce virtual cross-flow detouring as an addition to existing NC analyses in [21]. Detouring defines a new degree of freedom in the search for the best trade-off between length of analyzed tandems and flow aggregation. The main idea is that, if a cross-flow is detoured over (parts of) another flow's path, both can be treated by the analysis algorithm as an aggregate on a longer tandem. This principle is illustrated in Figure 3.6, where xf_2 is detoured over s_{01}. Despite the additional load at servers to be detoured over, this approach attains improved, valid delay bounds under certain conditions.

The addition of pessimism is not an entirely novel idea, and can be seen as a generalization of the FP property introduced in Section 3.2.1. Both detouring and FP aim at maximizing the use of the PMOO aggregation property of NC and thus tightening bounds.

Detouring creates a new trade-off that can beat the standard Pay Multiplexing Only Once Analysis (PMOOA) strategy. The longer tandem will be able to hold more data in transit (added pessimism), and the issues of PMOOA prevent introduction of dangerous optimism by making it lack the ability to distribute load in a better way than on the original tandem. In summary, it is key to virtually detour a flow over its cross-flows such that the PMOO property has an impact. Then, the analysis of a more pessimistic setting can indeed result in better output bounds



(a) Original network from [25] **(b)** After detouring of flow xf_2

Fig. 3.6: Cross-flow detouring known to benefit from a longer tandem analysis of xf_1

In larger networks with flows taking different paths, there may be several virtual detouring alternatives in order to allow for the PMOO principle's application to different sets of their cross-flows – the amount of alternatives certainly increases with the network size. This creates potential for a combinatorial explosion, a common problem already identified for FP in Section 3.2.1 and TMA in Section 3.2.2. To overcome this issue, a simple heuristic was proposed, which simply checks all in-links and detour over the one that has the most flows on it.

Numerical evaluation

We compare in this section the resulting end-to-end delays produced by PMOOA+Detouring and compare them against TMA and SFA. We aim to derive the best delay bound for every flow. There are no semantics assigned; the flow with the best improvement could be the most important one.

We first evaluate how many flows have their end-to-end delay bound matched or reduced with the use of detouring, compared to the other analyses:

$$delay_{PMOOA+Detouring} \leq delay_X \tag{3.1}$$

Results are presented in Figure 3.7. PMOOA+Detouring is able to match or improve delay bounds of TMA for at least 53.0 % of the analyzed flows – this lowest value is obtained in the largest network. In contrast to PMOOA without Detouring, which matches at most 26.7 % of the analyzed flows compared in the largest network, the addition of detouring is beneficial. As expected, the use of detouring has thus almost no impact on the existing relation to SFA and ULP delay bounds [24].

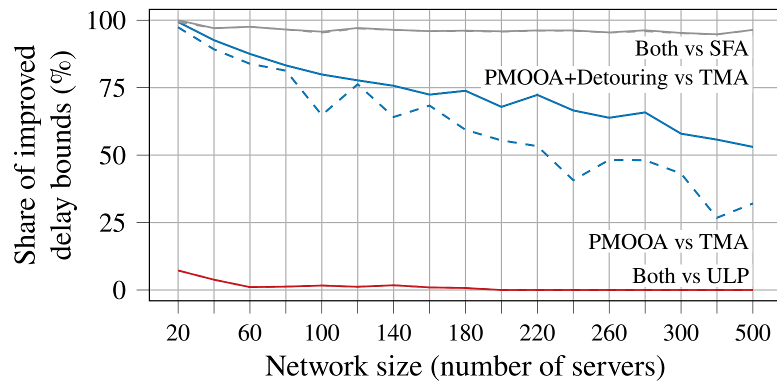


Fig. 3.7: Delay bound improvements over other network analyses

Conclusion

We show that virtual flow detouring is a powerful extension to the PMOOA. Resulting delay bounds are matching or even outperforming the state-of-the-art analyses that are considerably more involved such as TMA. Our contribution is based on the counter-intuitive idea of adding pessimism to the model, a generalization of the FP property. Due to a previously frowned upon characteristic of PMOOA, we can compute better delay bounds nonetheless.

3.4 Graph-based deep learning for speeding up formal verification

We noted in Sections 2.1, 3.2 and 3.3 that while some analysis methods provide impressive improvements regarding bound tightness, this gain in tightness is paid with a non-negligible cost in terms of execution time. To overcome this issue, we introduce in this section a powerful heuristic based on graph neural network (GNN). We follow the main concept introduced in Section 2.4. We show different use-cases of how this heuristic can be tailored to overcome pitfalls of some analyses, leading to fast and tight analyses.

We note that while the heuristics proposed here might not always give the best answer, they are integrated in the various NC analyses in such a way that the bounds produced are still formally valid. This is an important aspect of NC, since its strength is to produce formally valid bounds.

Including machine learning (ML) in formal methods is nothing new, as shown by the recent surveys from Amrani *et al.* [5] and Wang *et al.* [184]. Specifically for GNNs, they were used for prediction of satisfiability of Boolean satisfiability problem (SAT) by Selsam *et al.* [158], or basic logical reasoning tasks and program verification by Li *et al.* [119].

3.4.1 A heuristic for different NC analyses

Note This section is based on [69], published in *Proceedings of the 2018 International Workshop on Network Calculus and Applications*, 2018. The complete work is in Appendix A.1.4.

We reviewed in Sections 2.1 and 3.2 various DNC analyses, and noted, either via references to prior art or via numerical evaluations, that no single analysis outperforms all the others w.r.t. tightness depending on the network which is studied. This leads to a state where practitioners of DNC are left at their own judgment when deciding which analysis and NC properties to use when analyzing a given network in order to have a tight bound.

To illustrate this state of NC, we evaluate and compare TFA, SFA and PMOO, with different arrival bounding methods on a dataset of randomly generated networks in Figure 3.8. Additionally to the main analysis, different arrival bounding methods are also evaluated. We refer to the work from Bondorf and Schmitt [22] for a complete explanations of those arrival bounding methods.

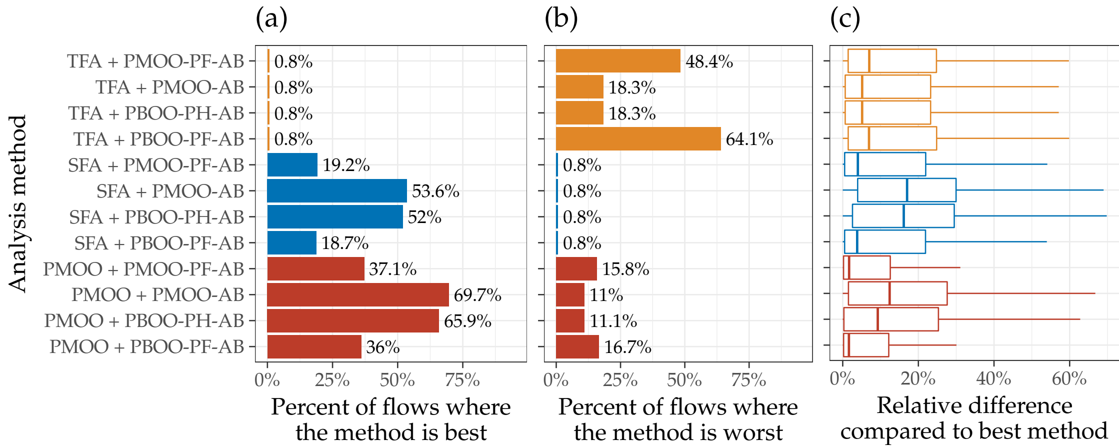


Fig. 3.8: Evaluation and comparison of the different network analysis methods against different metrics. *(a)* Ratio of flows where a given method produces the tightest bound compared to the other methods, *(b)* respectively the worst bound. *(c)* Relative difference between delay bound of a given method and the best method when the given method does not provide the best bound.

Figure 3.8 confirms that no single analysis is best, meaning that some level of expertise is required to understand which analysis to use given a network. To alleviate this requirement, we propose a simple algorithm illustrated in Algorithm 3.1. The goal is to appropriately select the best analysis method before doing the bound analysis via the *select_netcalc_method* function. We propose to use a GNN for performing this task.

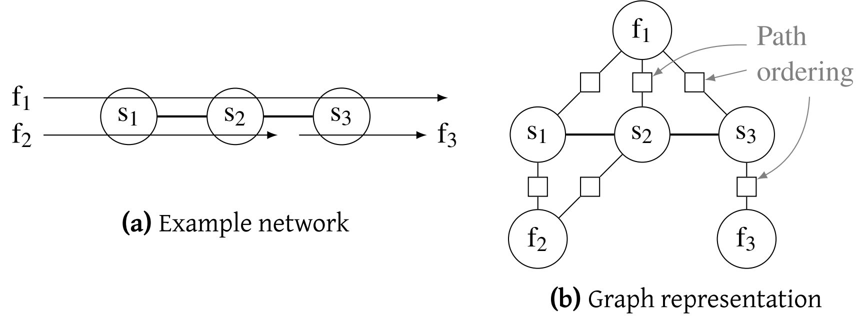
Algorithm 3.1 Adaptive network analysis

for all flow of interest $\mathcal{F}$ in network $\mathcal{N}$ **do**
  netcalc_method ← select_netcalc_method($\mathcal{N}, \mathcal{F}$)
  $bound_{\mathcal{F}}$ ← netcalc_method($\mathcal{N}, \mathcal{F}$)
end for

Graph transformation

In order to apply the concepts described in Section 2.2 to network calculus analysis and ultimately have an efficient *select_netcalc_method* function, we model the feed-forward server graph

and the flows crossing it into undirected graphs. Each server is represented as a node in the graph, with edges corresponding to the connections between servers. Each flow is represented as a node with edges connecting it to the path of traversed servers. Since the order of the servers which is traversed by a flow plays a large influence in network calculus, so-called *path ordering* nodes are added on the edges between the flow node and the server nodes. Figure 3.9b illustrates this graph encoding with the network from Section 3.4.1.



(a) Example network

(b) Graph representation

Fig. 3.9: Transformation from network calculus server graph to unidirected graph

Each node in the graph has the following input features:

- For server $\mathcal{S}$, we use the parameters of its rate-latency curve: $\mathbf{i}_{\mathcal{S}} = [rate_{\mathcal{S}}, latency_{\mathcal{S}}]$;
- For flow $\mathcal{F}$, we use the parameters of its token bucket curve: $\mathbf{i}_{\mathcal{F}} = [rate_{\mathcal{F}}, burst_{\mathcal{F}}]$;
- For a path ordering node $\mathcal{O}$, a categorical encoding of the hop index is used as input feature. We use standard one-hot encoding, namely $\mathbf{i}_{\mathcal{O}}$ is a vector with a one at the hop index, and zeros otherwise (e.g.: in Figure 3.9b, we have $\mathbf{i}_{\mathcal{O}}^{f_1-s_1} = [1,0,0]$, $\mathbf{i}_{\mathcal{O}}^{f_1-s_2} = [0,1,0]$, $\mathbf{i}_{\mathcal{O}}^{f_1-s_3} = [0,0,1]$).

For the output prediction of each node representing a flow $\mathcal{F}$, we wish to have a vector of end-to-end latency bound for the 12 methods presented earlier, namely:

$$\mathbf{o}_{\mathcal{F}} = [delayBound_{\mathcal{F}}^{m_1}, delayBound_{\mathcal{F}}^{m_2}, \ldots, delayBound_{\mathcal{F}}^{m_{12}}]$$

with m_x the different NC analysis methods. Similar output vectors may be used for the servers' backlog bound.

By this method, the GNN is an efficient approximation of the delay bounds of the different methods. By sorting the predicted delay bounds and taking the one resulting in the minimum, we build the *select_netcalc_method* function.

Numerical evaluation

We first assess in this section the precision of the predicted end-to-end latency bounds. Figure 3.10 illustrates the absolute relative difference between the predicted end-to-end delay bounds (*i.e.* the vector $\mathbf{o}_{\mathcal{F}}$) and the bound given by the analytical method (named here *ground truth*). The overall median value is 2.5 %, with larger errors in case of predicting the output of PMOO.

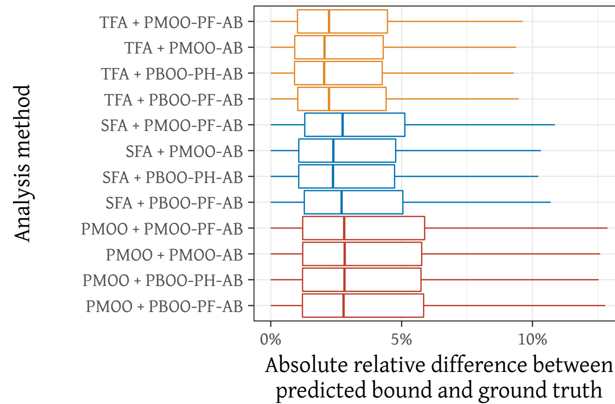


Fig. 3.10: Precision of the predicted end-to-end latency bound

Based on those results, we evaluate in Figure 3.11 the ability of Algorithm 3.1 to produce the tightest per-flow end-to-end delay bound. Different strategies are evaluated: the ML-based ones using the GNN, random choices, or simply taking the best method overall. The machine learning based strategies outperform all the other strategies in Figure 3.11, with the ability to produce the tightest result for 88 % of the studied flows for the *ML top 2* strategy, outperforming all the other evaluated strategies.

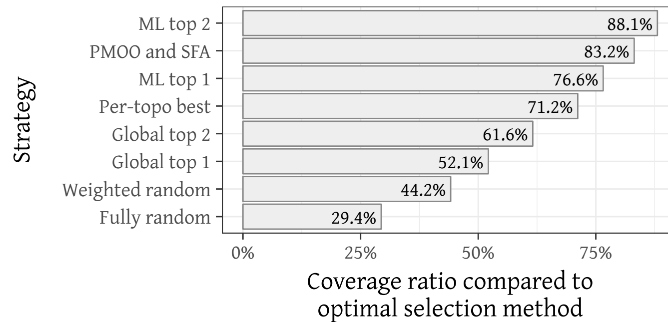


Fig. 3.11: Ability of a selection method to produce the tightest end-to-end delay bound

Conclusion

We contributed a novel heuristic for DNC using graph-based deep learning. Our approach is based on the application of GNN and a mapping from feed-forward server graphs and the flows crossing them to graphs which can be used for training a neural network. We showed via a numerical evaluation that our approach is able to reach good accuracy and predict which network analysis will produce the tightest bounds. Additional results in [68] show that this heuristic can be used at a small computational cost compared to traditional network analyzes.

3.4.2 Predicting best tandem decompositions with DeepTMA

Note This section is based on [62] published in *Proceedings of the 38th IEEE International Conference on Computer Communications*, 2019, [64] published in *Proceedings of the 25th IEEE Symposium on Computers and Communications*, 2020, and [65] published in *IEEE Transactions on Network Science and Engineering*, 2021. The complete works are in Appendices A.1.5 to A.1.7.

With our review of Tandem Matching Analysis (TMA) [24] in Section 3.2.2, we identified its major pitfall: in order to find the best tandem decomposition, TMA needs to evaluate all possible tandem decompositions via an exhaustive search. On a single tandem with n servers, this amounts to a total of 2^{n-1} potential decompositions to explore, leading to hours of computations on larger networks [24].

We propose in [62, 64, 65] and in this section to avoid this exhaustive search by applying the lessons learned from [69] and Section 3.4.1, namely: use a GNN to predict the most promising tandem decomposition(s), and perform the TMA only on this subset of decompositions. Our approach – called *DeepTMA* – achieve considerably faster execution times than TMA without considerably compromising on delay bound tightness.

The main intuition is to transform the NC server graph and flows into an undirected graph. This graph representation is then used as input for a neural network architecture able to process general graphs, which will then predict the tandem decomposition resulting in the best residual service curves. Our approach is illustrated in Figure 3.12. Since the delay bounds are still computed using the formal network calculus analysis, they inherit their provable correctness.

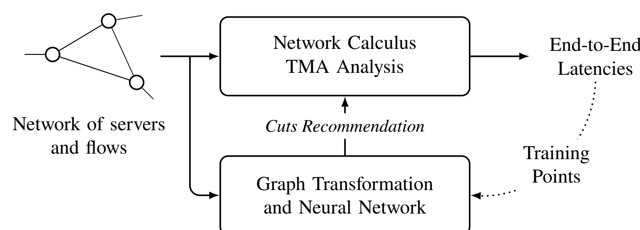
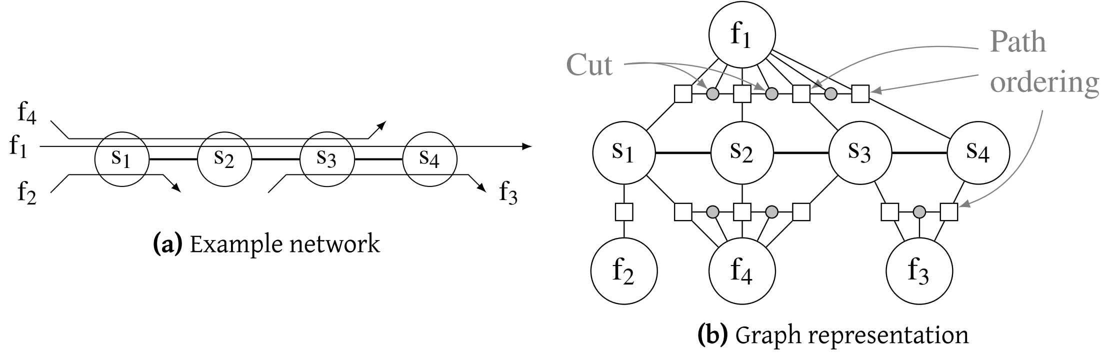


Fig. 3.12: Overview of DeepTMA and its integration within NC

Graph transformation

We model NC's directed network as an undirected graph, following a similar process than in [69] and Section 3.4.1. Figure 3.13b illustrates this graph encoding on the network from Figure 3.13a.

We start with the graph transformation explained in Section 3.4.1, namely each server and each flow is represented as a node in the graph. A flow's path is represented using edges, with the path ordering nodes explained in Section 3.4.1. This path property is especially important in the TMA since the order, and hence position of cuts, has a large impact on dependency structures. In order to represent the TMA cuts introduced by Definition 3.1 in Section 3.2.2, each potential cut between pairs of servers on the path traversed by the flow is represented as an additional node. This *cut node* is connected via edges to the flow and to the pair of servers it



(a) Example network

(b) Graph representation

Fig. 3.13: DeepTMA graph transformation

is associated to. We use the same node features introduced in Section 3.4.1 for the server, flow and path ordering nodes. Neither cut nodes nor edges have input features.

Based on this description of the server graph, the problem of choosing the best tandem decomposition to give to the NC analysis is formulated as a classification problem. Each cut node has to be classified in two classes: perform a cut between the pair of servers it is connected to or not: $[cut, \overline{cut}]$. The overall prediction to be fed back, i.e., the selection of one out of TMA's potential decompositions for a given foi's path, is defined by the set of all cut classifications for this path.

Numerical evaluation

Figures 3.14 and 3.15 illustrate benchmarking results of DeepTMA. We compare against TMA and the established SFA and PMOO heuristics of NC. These heuristics greedily decide on a single contention model, ignoring arrival and service curves.

To evaluate the delay bound tightness, we use the relative error to TMA:

$$relative\ error_{\mathrm{foi}} = \frac{delay_{\mathrm{foi}}^{\mathrm{heuristic}} - delay_{\mathrm{foi}}^{\mathrm{TMA}}}{delay_{\mathrm{foi}}^{\mathrm{TMA}}} \tag{3.2}$$

A value of $relative\ error_{\mathrm{foi}}$ close to zero indicates that the heuristic produced a tight result compared to the exhaustive search. Larger values indicate that the heuristic chose a tandem decomposition.

All heuristics outperform a consistent worst choice of contention models as shown in Figure 3.14. DeepTMA-derived delay bounds are tightest among these heuristics, deviating from TMA by no more than 6 % in our experiments from [62].

In terms of execution time, DeepTMA is minimally slower than PMOO but faster than SFA and TMA, as illustrated in Figure 3.15. Moreover, recent work from Scheffler *et al.* [149] showed that the TMA cannot be parallelized easily and a speedup of only one order of magnitude was observed.

We also compared DeepTMA against various heuristics in [62] and [65]. In Figure 3.16, we compare our GNN heuristic against a random heuristic. This heuristic randomly samples a small part of TMA's search space per tandem in the analysis, with the n in RND_n representing the size of the sample. DeepTMA is able to achieve much better results than this random

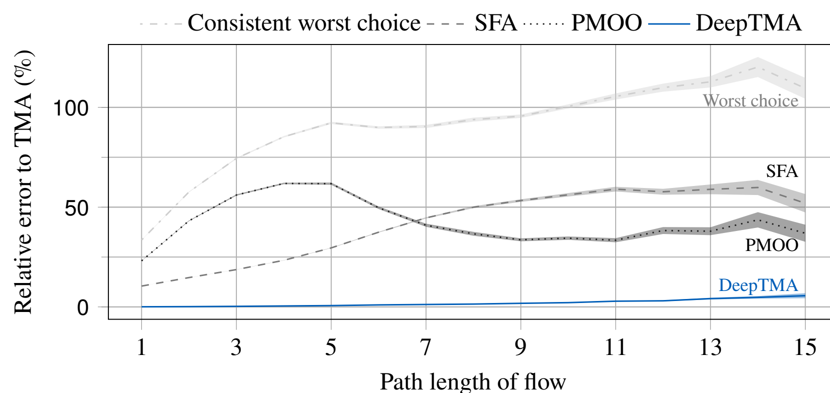


Fig. 3.14: Comparison of DeepTMA to existing NC heuristics with respect to relative error

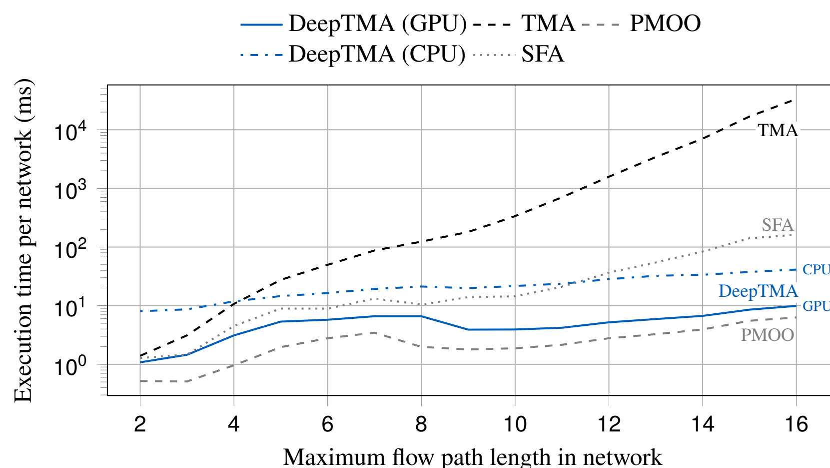


Fig. 3.15: Comparison of DeepTMA to existing NC heuristics with respect to execution time

heuristic in Figure 3.16. We also compared DeepTMA against NC-based heuristics in [62] and simpler ML-based heuristics in [65]. Our numerical evaluation show that DeepTMA is consistently better than all the heuristics which were evaluated.

Conclusion

We contribute in [62, 64, 65] a new framework that deeply combines network calculus and deep learning for producing tight and efficient bounds. It solves the main bottleneck of the existing TMA, namely its exponential execution time growth with network size, by using predictions for effectively selecting the contention models in the network calculus analysis.

Via a numerical evaluation, we show that our heuristic is accurate and produces end-to-end bounds which are almost as tight as TMA. DeepTMA is as fast as or faster than previously widespread methods – namely SFA and PMOO – even when analyzing larger networks with up to 14 504 flows in [65], and with a gain in tightness exceeding 50 % in some cases.

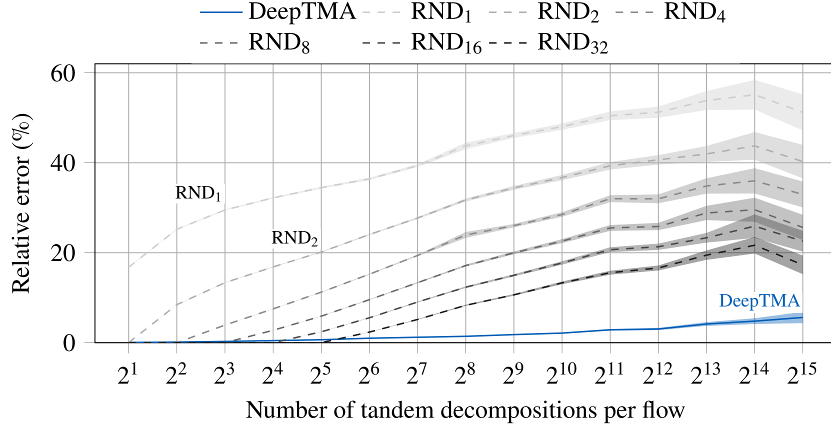


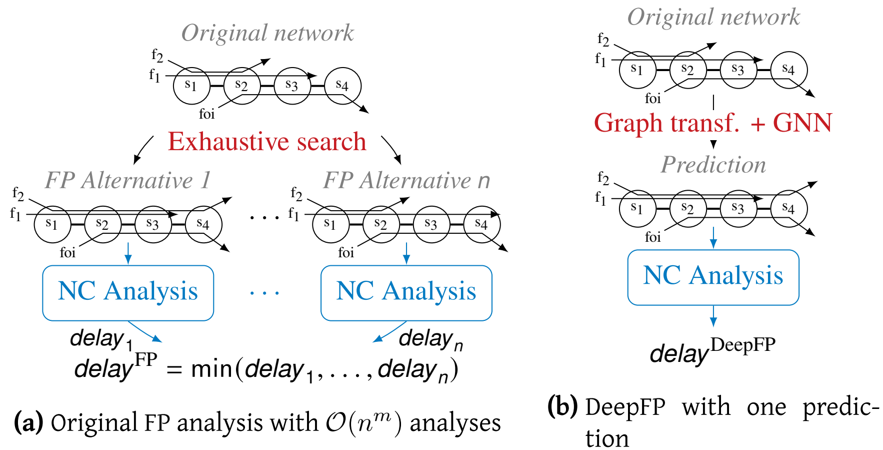
Fig. 3.16: Relative error of DeepTMA against a random heuristic

3.4.3 Predicting best flow prolongations with DeepFP

Note This section is based on [75], published in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2021.

With our review of Flow Prolongation (FP) [18] in Section 3.2.1, we identified a second network analysis method from the NC literature with good tightness but suffering from poor scalability due to its reliance on an exhaustive search. On a single tandem with n servers and m cross-flows, the analysis time of FP grows in $\mathcal{O}(n^m)$ potential flow prolongations to explore.

We propose here *DeepFP*, a method for avoiding FP's exhaustive and bringing efficient FP to NC based on the lessons learned from Sections 3.4.1 and 3.4.2. Overall, the main concept behind DeepFP is similar to DeepTMA. To overcome exhaustive searches in the algebraic FP NC analysis, we use the prediction of a GNN to restrict it to a small subset of best potential flow prolongations. This concept is illustrated in Figure 3.17.



(a) Original FP analysis with $\mathcal{O}(n^m)$ analyses

(b) DeepFP with one prediction

Fig. 3.17: Overview of FP and DeepFP

By demonstrating that we can make the FP analysis scale this way, we also reveal that its

impact on the derived delay bound is very sensitive to the network model's assumptions. The foremost contribution of DeepFP in [75] is the FP analysis of FIFO networks. Under this assumption and applying the state-of-the-art algebraic Least Upper Delay Bound (LUDB) analysis [15, 17] and its tool Delay Bound Rating AlgoritHm (DEBORAH) [16], we derive entirely new conditions for beneficial flow prolongations, train the GNN and acquire significantly improved delay bounds.

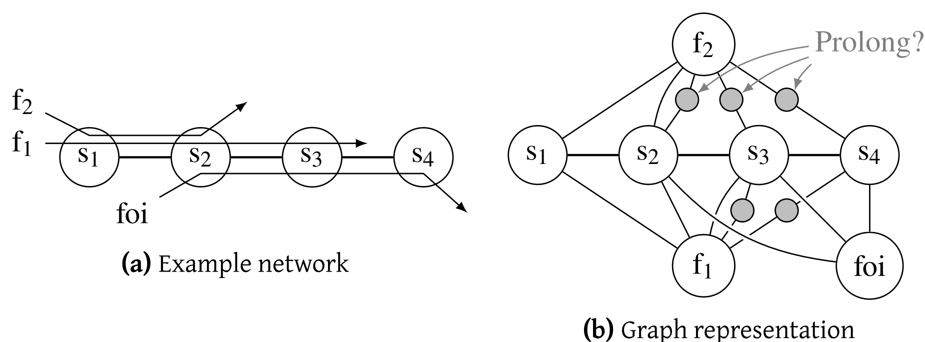
Graph transformation

We follow a similar graph transformation than in Sections 3.4.1 and 3.4.2, as illustrated and applied in Figure 3.18b on the network from Figure 3.18a. Each server and each flow is represented as a node in the graph, with the features already introduced in Section 3.4.1. Flows' path are encoded using edges in the graph. Additionally, the foi receives an extra feature representing the fact that it is the analyzed flow.

Compared to the original DeepTMA graph model, we simplify one aspect: we do not include path ordering nodes that tell us the order of servers on a crossed tandem. DeepTMA was shown to benefit only marginally from the effort to incorporate this additional information [64] and we confirmed the same behavior in preliminary DeepFP numerical evaluations.

To represent the flow prolongations, *prolongation* nodes ($P_{f_i}^{s_j}$) connecting the cross-flows to their potential prolongation sinks are added to the graph. Those nodes contain the hop count according to the foi's path as main feature – this is sufficient to later feed the prolongation into the NC analysis, path ordering nodes are not required for this step either.

The last server of a cross-flow's unprolonged path is also represented as a node (s_3 for f_1 and s_2 for f_2 in Figure 3.18b). Those nodes represent the choice to not prolong a flow.



(a) Example network

(b) Graph representation

Fig. 3.18: DeepFP graph transformation

Numerical evaluation

We compare DeepFP against PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$, modified analyses where the FP property is only applied to the foi and not the also in the different recursions required during the NC analysis.

To quantitatively evaluate the performance of our approach, we use the relative gap between the delay bound given by PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$ and the delay bound given by

a heuristic, incl. the non-FP original analysis:

$$delay\ bound\ gap_{\text{foi}}^{\text{FP}} = \frac{delay_{\text{foi}}^{\text{heuristic}} - delay_{\text{foi}}^{\text{FP}_{\text{foi}}}}{delay_{\text{foi}}^{\text{FP}_{\text{foi}}}} \tag{3.3}$$

A value of $delay\ bound\ gap_{\text{foi}}^{\text{FP}}$ close to zero indicates that the heuristic produced a tight result compared to the exhaustive search. Larger values indicate that the heuristic chose a bad prolongation, i.e. the bound is loose.

The results are shown in Figure 3.19. First to note is that FP does not have a significant impact in PMOO – we confirm the finding of Bondorf [18] in a larger evaluation by observing an average gap between PMOO-FP$_{\text{foi}}$ and PMOO of just 3.7 %. Neither the random heuristic nor DeepFP can thus achieve a considerable delay bound improvement, although the predictions taken are very accurate.

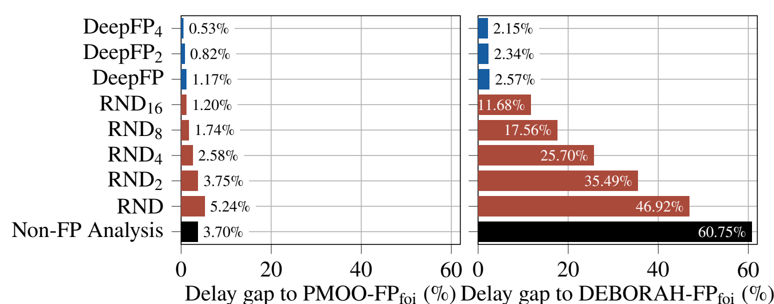


Fig. 3.19: Average delay bound gap of heuristics against PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$

For DEBORAH-FP, we can report a completely different picture. Having brought the FP property to the DEBORAH analysis had a huge impact on the delay bound tightness. We see that an average gap of 60.75 % between DEBORAH and DEBORAH-FP$_{\text{foi}}$ analysis results was opened when adding the exhaustive FP$_{\text{foi}}$ feature. Moreover, reducing the effort by random selection of prolongation alternatives did not perform well, even RND$_{16}$ leaves an average gap of 11.68 %. On the other hand, DeepFP closes this gap successfully. Even the version with a single prediction pushes the gap down to 2.57 % such that an increase of proposed prolongation alternatives does not have a big impact anymore.

To evaluate the scalability of our approach with respect to the network size, we also evaluated DeepFP on networks with a larger number of servers (up to 16) and flows (up to 256). Numerical results are summarized in Figure 3.20. As in Equation (3.3), we define the delay bound gap to PMOO and DEBORAH (i.e. the analyses without the FP property) as:

$$delay\ bound\ gap_{\text{foi}}^{\text{non-FP}} = \frac{delay_{\text{foi}}^{\text{non-FP}} - delay_{\text{foi}}^{\text{heuristic}}}{delay_{\text{foi}}^{\text{non-FP}}} \tag{3.4}$$

For the PMOO analysis, the random heuristic results in a negative delay bound gap in average, namely the resulting delay bounds are worse than by simply using the standard PMOO analysis, even for the larger values of $k = 32$. Despite this, DeepFP is able to achieve an average gain in tightness of 1.06 % for PMOO. For the DEBORAH analysis, the random heuristic results in a gain in tightness of only 0.25 %, where DeepFP is able to achieve a gain of 13.74 %.

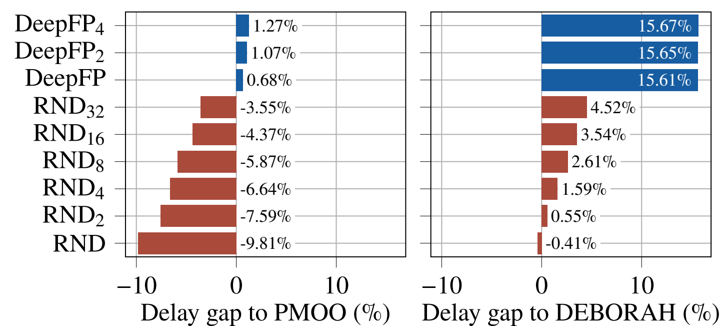


Fig. 3.20: Average delay bound gap of DeepFP to standard PMOO and DEBORAH on the larger networks

We first illustrate the average relative execution time of the FP analyses against the non-FP analysis in Figure 3.21, namely:

$$\frac{\textit{Execution time FP}}{\textit{Execution time non-FP}} \tag{3.5}$$

This measure helps us understand the cost of using FP. In average, DeepFP with Graphic Processing Unit (GPU) acceleration is approximately an order of magnitude faster than PMOO-FP_{foi}, and almost three orders of magnitude faster than DEBORAH-FP_{foi}. Taking into account the tightness of the method illustrated earlier in Figure 3.19, those results show that DeepFP is able to achieve a good balance between tightness and computational cost.

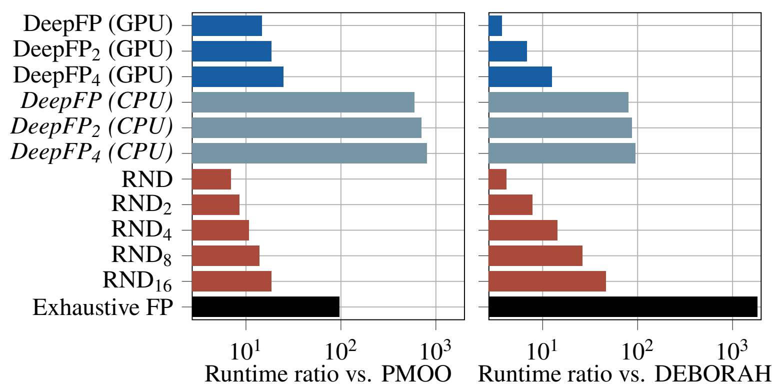


Fig. 3.21: Average relative execution time of different analyses

Conclusion

With DeepFP, we introduced an approach for making the NC analysis feature FP scale. FP can be paired with either of the two predominant flow multiplexing assumptions, arbitrary or FIFO, and we show that it is most impactful when bounding the flow of interest's delay.

Our results are especially relevant to any system designed around FIFO-multiplexing and -forwarding of data. Most notable are Ethernet-based networks like AFDX, AVB or TSN. Even though they follow the "FIFO per priority queue" design, their NC model is essentially a FIFO

system model, as illustrated for example by Boyer and Fraboul [33]. Our results can be combined with existing works on service modeling of the specific schedulers used in those systems.

When scaling to larger networks, where the existing PMOO-FP was known to struggle with computational effort, DeepFP still works. Without considerable loss of prediction accuracy we gain delay bound tightness of 1.06 % compared to standard PMOO, and 13.7 % compared to DEBORAH.

3.5 Conclusion on network calculus improvements

We contributed in this chapter various works and improvements on the analysis methods of DNC. One conclusion from our review of the literature on DNC is that while various analysis methods have been proposed, there is no clear winner on which one is the best in all possible scenarios and networks. With all these methods there is also a strong correlation between gains in tightness and execution time as illustrated earlier in Figure 1.1.

This means that a practitioner of DNC has to use expert knowledge in order to decide which method is the most appropriate given a network to analyze. One major contribution presented in Section 3.4 is the introduction of a fast and efficient heuristic for DNC based on GNN to alleviate the need for this expert knowledge. With our heuristic, we are the first ones to contribute tight and efficient methods. Our methods are both competitive in terms of computational efforts with early works in DNC such as SFA and PMOOA, and competitive in terms of tightness with more recent and involved methods such as TMA and FP.

With our works, we also contributed openly accessible NC-specific datasets (see Appendix B). This is also an important step for the NC community, since there is currently a lack of common datasets for evaluating and comparing the results of analyses.

Finally, our works also illustrates that data-driven methods open the door to an ideal fast and tight method as illustrated in Section 3.4. Our GNN-based approach may also be extended to adjacent problems such as DNC-based network optimization or network design. An example of this is that our research on GNN and NC sparked interest in the NC community, as illustrated by the work from Mai and Navet in [122] and [123], which looked at applications of GNNs for the configuration of TSN networks.

4. APPLICATION OF GRAPH-BASED DEEP LEARNING METHODS FOR COMPUTER NETWORKS

4.1 Introduction . . . 37
4.2 Related work . . . 37
4.2.1 ML for performance evaluation . . . 38
4.2.2 ML for routing . . . 38
4.2.3 ML for network protocols . . . 38
4.3 Graph models for performance evaluation with DeepComNet . . . 39
4.4 Graph models for generating network protocols . . . 42
4.5 Graph models for reasoning about network protocols . . . 45
4.6 Conclusion on machine learning methods for computer networks . . . 49

4.1 Introduction

In Section 2.4 we introduced a novel concept for solving some issues in networking, namely use graph transformations in combination with graph neural networks (GNNs) for leveraging patterns in network problems and follow a data-driven approach for addressing them. While related machine learning (ML) approaches have been used in various works, they usually require fine tuning and expert knowledge for designing input features. Traditional approaches may also not cope with network of various sizes, an aspect easily handled using GNNs.

While we investigated in Chapter 3 – and more specifically Section 3.4 – how it can be applied to deterministic network calculus (DNC), we explore in this chapter other areas where GNNs are relevant: *(i)* performance evaluation in Section 4.3, *(ii)* network protocol design in Section 4.4, and *(iii)* validation and synthesis of network configuration in Section 4.5.

This chapter helps strengthening the versatility of our approach. We illustrate that by correctly incorporating knowledge and relationship about the tackled problem in the data graph, GNNs can address a large variety of tasks.

4.2 Related work

ML as attracted a lot of attention in the networking community in the last years, as shown in recent surveys from Wang *et al.* [186], Usama *et al.* [179], Xie *et al.* [192], and Zhang *et al.* [197]. We review here specific works related to the problems addressed in this chapter.

4.2.1 ML for performance evaluation

ML is increasingly been used for performance modeling. For instance, Tian and Liu [175] applied the Support Vector Regression (SVR) model of Transmission Control Protocol (TCP) bandwidth prediction application from Mirza *et al.* [133] to improve Quality-of-Service (QoS) of media streaming over Hypertext Transfer Protocol (HTTP). Tariq *et al.* [169] proposed What-If Scenario Evaluator (WISE), a framework for evaluating architecture changes in communication networks using Causal Bayesian Networks (CBN). Hours *et al.* [86] used Bayesian causal inference for modeling the bandwidth of TCP flows.

Graph models has also recently attracted various works. In the scope of wireless networks, GNNs were used by Lee *et al.* [117] for link scheduling, Shen *et al.* [161] for interference channel power control, and Nakashima *et al.* [137] for wireless channel allocation.

4.2.2 ML for routing

The question of distributed routing protocols based on machine learning has already attracted various researchers. Early work on this topic include *Q-Routing* from Boyan and Littman [31], *COllective INtelligence* (COIN) from Wolpert *et al.* [190], or *distributed Gradient Ascent Policy Search* (GAPS) from Peshkin and Savova [141]. Their general approach is to use multi-agent reinforcement learning in combination with a network-wide utility function. More recently, Valadarsky *et al.* [180] also proposed to use reinforcement learning, with the goal of using past traffic matrices in order to guide route calculations. Compared to those works, we use here semi-supervised learning in order to more easily specify the routing policy which is expected. Previous work also often predetermined or constrained the specification and format of the communication, whereas our approach leaves the content or format of the exchanged information as a parameter to be learned. Our work also evaluates key aspects of routing protocols, namely resilience against packet loss and inclusion of network dynamics.

A supervised learning approach was recently proposed by Mao *et al.* [125] using Supervised Deep Belief Architectures, with a focus on speed of route computation. Compared to their approach, our method can be applied to a wider range of network topologies since it is independent of the underlying structure of the topology.

The challenge of training agents to communicate and realize a common goal has attracted work in other domains. Foerster *et al.* [56] applied *Deep Distributed Recurrent Q-Networks* (DDRQN) for solving logic riddles. Sukhbaatar *et al.* [167] proposed a deep neural network architecture called *CommNet* for developing communication between agents on the task of multi-turn games, traffic junction or logic riddles. In both approaches, no constraint on communication structure is enforced as a broadcast channel is used.

4.2.3 ML for network protocols

Additionally to the proposal for bringing ML to routing protocols, ML was also used in other various areas of network protocols. We review here some of the works addressing some challenges in network protocol design and functions with data-driven approaches.

In the scope of congestion control, various works were performed to improve the bandwidth utilization while reducing latency. Winstein and Balakrishnan [189] introduced RemyCC,

one of the early work on data-driven approach to congestion control. Neural network (NN)-based congestion control were investigated by various works, such as Muses from Wang *et al.* [187], Performance-oriented Congestion Control (PCC) Vivace from Dong *et al.* [51], and Indigo from Jay *et al.* [93, 94].

Finally for adaptive video streaming, solutions to optimize users' Quality-of-Experience (QoE) based on adaptive bitrate were also proposed. Mao *et al.* [126] used reinforcement learning to dynamically adapt video bitrate. More recently, Yan *et al.* [194] proposed Fugu for adaptive bitrate, based on a hybrid model predictive control (MPC) and NN approach.

4.3 Graph models for performance evaluation with DeepComNet

Note This section is based on [60] published in *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, 2017, and [61] published in *Performance Evaluation*, 2019. The complete works are in Appendices A.2.1 and A.2.2.

Network and traffic models are important tools in order to predict how a given network architecture will behave. This is an important task for architecture design and QoS in an increasing number of applications. Different techniques have been developed for this purpose, such as mathematical modeling, simulations or measurements. While those techniques are usually accurate, they often require precise measurements of key performance indicators such as round-trip time (RTT) or loss probability, and are often tailored to specific parts or flavors of the studied network protocols. Early ML-based models for performance evaluation such as [133, 175, 169, 86] – while producing good accuracy – still require the same features such as RTT or loss probability.

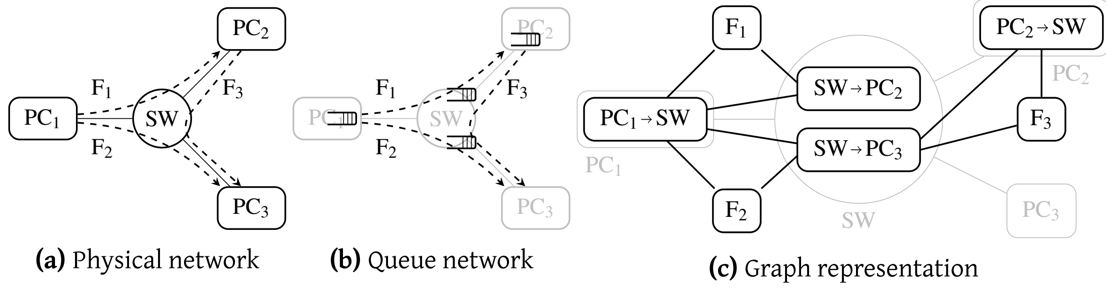
Our main contribution in [60, 61] is a performance model which only uses the studied network topology and flows description as input in order to predict bandwidth utilizations and latencies of flows interacting on this network. For this purpose, we apply the concepts introduced in Section 2.4. The intuition behind our approach is to map network topologies and flows to graphs, and then train GNNs on those graphs. This enables us to avoid the task of engineering high-level protocol-specific input features such as RTT, which usually require expect knowledge on the network protocol which is modeled.

Graph transformation

The main intuition for our graph transformation is to use the queuing network as modeling graph, with additional nodes representing the flows in this network. An illustration of this queuing network is given in Figure 4.1b, which is the queuing representation of the example network illustrated in Figure 4.1a with one switch or router interconnecting three PCs with three flows.

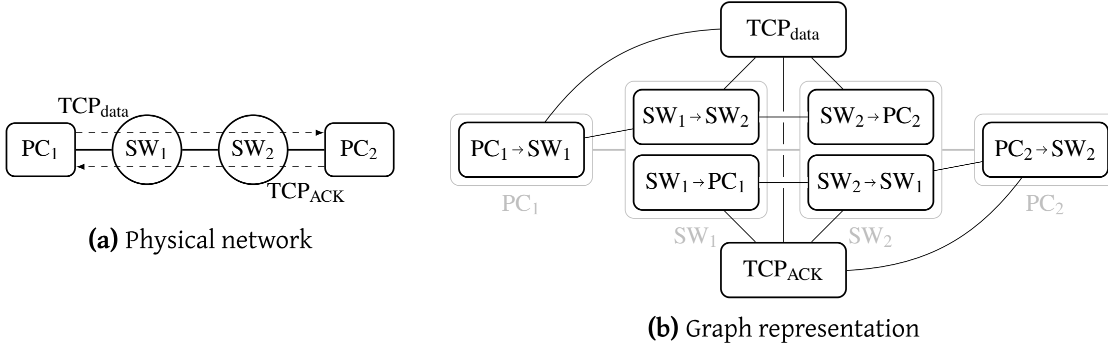
If the studied protocols are bidirectional, *i.e.* packets are sent in both directions between the sender and received, both directions need to be included during the graph transformation. This is relevant in protocols with acknowledgment packets such as TCP for example. An illustration of those additional queues is given in Figure 4.2.

Based on this queue network, the graph transformation is as follows. Nodes in the graph



(a) Physical network **(b)** Queue network **(c)** Graph representation

Fig. 4.1: DeepComNet graph transformation



(a) Physical network **(b)** Graph representation

Fig. 4.2: DeepComNet graph transformation for TCP

correspond to the queues traversed by the flows in the network topology as well as specific nodes representing the flows. For the node features l_v, a vector encoding the node type (*i.e.* if a node represents a flow or a queue) with one-hot encoding is used. Namely l_v is a vector with two values, with $[1, 0]^T$ is for queue nodes, and $[0, 1]^T$ for flow nodes. For simplification purpose, we assume here that every PCs and switches or routers to have the same behavior and all links in the topology to have the same capacity and latency. Additional features for distinguishing between different behaviors or node types may be used in case different configurations, types or link capacities are used.

Edges connect the queues which are used by the flows according to the physical topology of the network. Additionally, edges between the flows and their traversed queues are used in order to model which path is traversed by the flows. Figures 4.1c and 4.2b are examples of our graph transformation applied to the topologies from Figures 4.1a and 4.2a respectively.

For each flow node in graph, the ML task is then to predict the performance indicator which is required. In the case of latency or bandwidth prediction, this becomes a regression task. A supervised approach is used for training the GNN.

Numerical evaluation

For the numerical evaluation of our approach in [61], we focused on two use-cases. First we evaluate the capabilities of our approach at predicting the steady-state bandwidth of TCP flows sharing different bottlenecks. For building our dataset, the ns2 simulator was used with the Reno TCP flavor [91] on daisy-chain topologies with randomly assigned number of nodes and flows.

Secondly, we evaluate the capabilities of our approach at predicting the end-to-end latencies of User Datagram Protocol (UDP) flows with constant bitrates sharing different bottlenecks. We follow the same approach as for the previous use-case, namely multiple random topologies with UDP flows are generated and evaluated using simulations.

Results are presented in Figure 4.3. Various GNN models were used, based on the gated graph neural network (GGNN) model from Li *et al.* [119] and presented in Section 2.2. We evaluated different variants for replacing The Gated Recurrent Unit (GRU) cell present in the GGNN model, namely using a Recurrent Neural Network (RNN) [144, 188] and a Long Short-Term Memory (LSTM) [85]. For the TCP use-case, we also additionally evaluated a SVR model similar to the one from [134] using the RTT and loss probability as input, and a feed-forward neural network (FFNN) model using the same input features as the SVR.

The median relative absolute error of 11.6 % for the SVR model, which confirms the 10 % median relative error from Misra *et al.* [134]. The FFNN provides better results, with a median relative error of 3.5 %. Those values are used as a baseline for comparison purposes.

Regarding the GGNN models, all architectures evaluated here are able to predict the TCP bandwidths with a median relative error below 1 %, outperforming the values from the SVR by one order of magnitude, and also outperforming the FFNN using high-level input features. This highlights our main motivation for using GGNNs.

Regarding UDP latencies, we are also able to reach a median relative error below 1 %. The LSTM-based GGNN architectures provide better results on the TCP use-case, while the GRU-based ones are more suited to the UDP use-case. We note that using stacked memory cells for the GGNN enables better accuracy for the TCP use-case, while not providing better results in the UDP use-case.

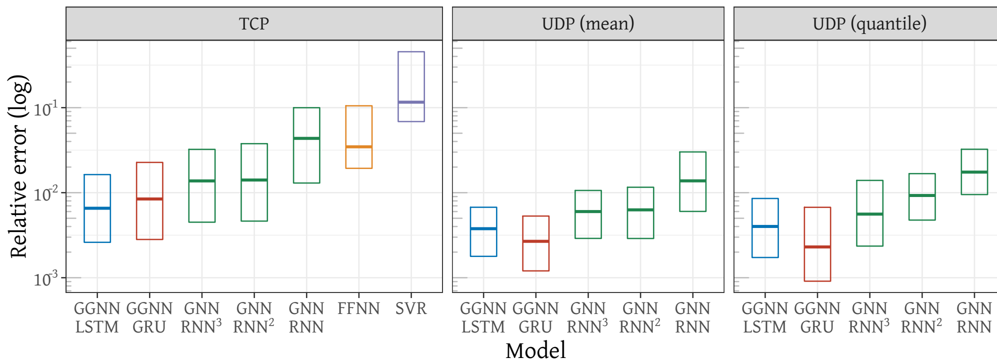


Fig. 4.3: Comparison of the evaluated machine learning methods according to the relative error. Bars represent respectively the 25, 50 and 75 percentiles.

Conclusion

In [60, 61], we contributed one of first models applying GNNs to the performance evaluation in networking. Overall, our results and additional results in [60, 61] illustrate that GNNs can efficiently predict key performances of flows in networks based only a representation of the network topology. Similar GNN-based models were later used for similar performance evaluation, such as the work from Rusek *et al.* in [145], from Andreoletti *et al.* [7] and from Suzuki *et al.* [168].

4.4 Graph models for generating network protocols

Note This section is based on [68] published in *Proceedings of the 2018 SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, 2018. The complete work is in Appendix A.2.4.

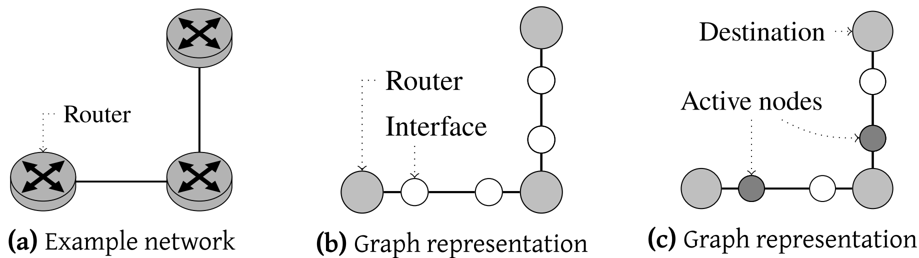
We propose in this section and [68] to investigate the question of automatic network protocol design using recent methods from deep learning. Our contribution is a novel approach for training a network of independent agents such that they cooperatively exchange information and solve a common goal in a fully distributed manner without central control. We address more specifically the question of distributed routing protocols. From a network protocol perspective, the routing agents should autonomously develop a network protocol akin to Routing Information Protocol (RIP) or Open Shortest Path First (OSPF), *i.e.* exchange topology information and perform local path computations based on the exchanged information. Traditional properties from routing protocols are also considered, namely handling topology changes and packet losses.

With our approach we also contribute a novel extension of GNNs, which we name Graph-Query Neural Network (GQNN). We evaluate our approach on various topologies from real networks from Knight *et al.* [104] and show that our approach leads to the creation of communication protocols able to exchange data about topology information as well as topology changes.

We are interested in training NNs on two important aspects of network protocols. The first one is the protocol itself, namely how to distribute topology information among different nodes. The second one is how to compute routes given a topology and link weights. We will follow the approach from Section 2.4, namely transform network topologies and the attributes specific to routing into graph, and process them using a GNN architecture.

Graph transformation

The main intuition behind the input feature modeling is to use the topology as input graph $\mathcal{G}$, with additional nodes representing the network interfaces as illustrated in Figure 4.4. In order to enforce communication between nodes according to the physical network topology, no additional edge is added to the graph. As for traditional routing protocols, each router in the topology is assigned an integer identifier, noted R_i. Nodes representing routers in the graph use this identifier encoded as a one-hot vector for their initial representation $\mathbf{i}_v$. Nodes representing interfaces use a weight parameter (*e.g.* based on the link bandwidth) for $\mathbf{i}_v$.



(a) Example network **(b)** Graph representation **(c)** Graph representation

Fig. 4.4: Routing graph transformation

Graph-Query Neural Network (GQNN)

As opposed to all the other works using GNNs presented in Chapters 3 and 4 which used a fairly unmodified GGNN architecture, we extended the GGNN with a mechanism for querying specific information for a prediction, namely a route given a destination address. Our aim is to bring a mechanism akin to the Scaled Dot-Product Attention from Vaswani *et al.* [181] to our architecture.

We explore here the local computation of the routing table based on the topology information which was distributed by the different nodes in the graph. Given a destination router identifier R_d, each router must locally decide which output interface should be used. Based on the graph transformation from the previous section and a given algorithm for path calculation for supervised learning, this is modeled by labeling the interfaces with $\mathbf{o}_i = [1]$ if they are used for transmitting packets to router R_d, and $[0]$ otherwise, as illustrated in Figure 4.4c.

Our new GGNN architecture is illustrated in Figure 4.5. The hidden node representations $\mathbf{h}_v^{(t)}$ correspond to the messages which are transferred between nodes. Once the message passing is finished, each node in the graph has a local representation of the network topology $\mathbf{h}_v^{(T)}$. For determining which interface to use for a given destination router R_d, each router applies the following procedure on each of its interface nodes:

$$\mathbf{q}_d = Q\left(R_d\right) \qquad \text{query vector computation} \qquad (4.1)$$

$$\mathbf{o}_v = g\left(\mathbf{q}_d \odot \mathbf{h}_v^{(T)}\right) \qquad \text{output label as in Equation (2.10)} \qquad (4.2)$$

with $Q(\cdot)$ and $g(\cdot)$ FFNNs.

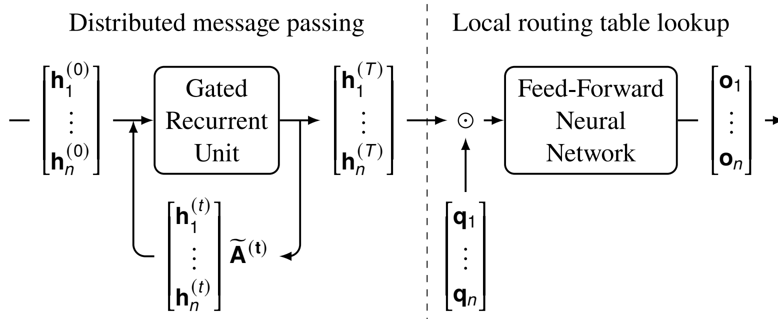


Fig. 4.5: Proposed Graph-Query Neural Network (GQNN) architecture

Numerical evaluation

For our numerical evaluation, we train our GNNs to follow two different algorithms for route calculation: *shortest path* and *max-min routing*. For shortest path routing, the GNN is trained against path calculations based on Dijkstra's algorithm, where each link is associated with a weight. For max-min fair routing [136], we aim at maximizing the minimum allocated bandwidth between all possible source-destination pairs in the network. Such routing strategy should lead to network topologies with less link overload than shortest path routing.

We evaluated our approach on real network topologies from the *Internet Topology Zoo* [104]. Since routing protocols are designed to run continuously and handle topology changes, we

define here two phases of the protocol: *cold-start* when the routing protocol is first initialized on the active routers, and *warm-start* when a node fails or a new node joins a network where the routing protocol already ran for some iterations.

Figure 4.6 illustrates the accuracy of the computed routes according to the two use-cases and the two phases. For a given topology, we define the accuracy as 1 if the route for a given destination is correct for all routers in the topology, and 0 otherwise. In average, accuracies of 98 %, respectively 95 %, could be reached for shortest path routing, respectively min-max routing. The learned protocol is able to better predict shortest path routing, where a perfect accuracy is reached for than 50 % of the evaluated topologies.

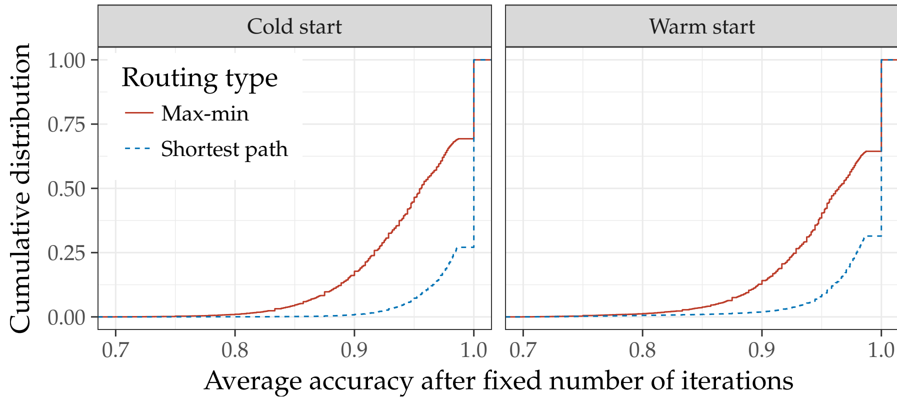


Fig. 4.6: Overview over the accuracy of the predicted routes

To assess the convergence time of the developed protocols, we evaluate the accuracy of the routing at different iterations of the fixed point evaluation of the GNN in cold-start and warm-start phases. The numerical results are presented in Figure 4.7. In case of topology changes (*i.e.* warm start), better accuracies are reached faster as routes only need partial reconfiguration. This shows that the protocol is indeed able to efficiently cope with and react to topology changes.

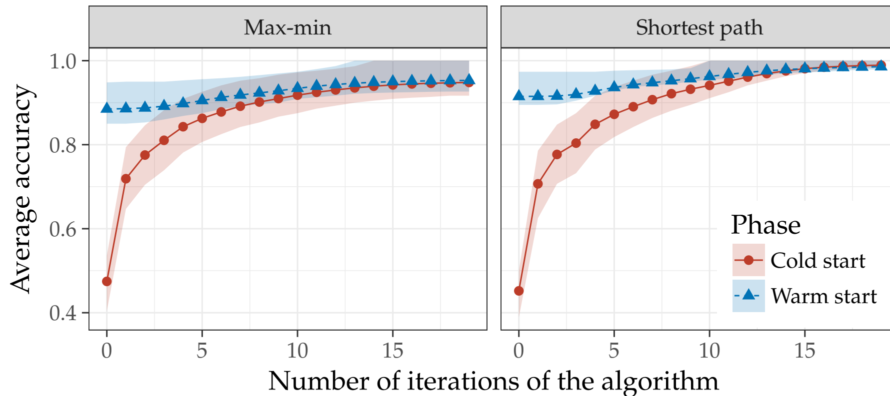


Fig. 4.7: Accuracy according to the iterations of the protocols. Areas indicate the 25 % and 75 % percentiles.

Conclusion

We contributed in this section a novel approach for automatic network protocol design using graph-based deep learning. Our method is based on our proposed extension of GNNs called Graph-Query Neural Network (GQNN) and a mapping from network topologies to graphs with special nodes representing network interfaces.

Shortest path and max-min routing were evaluated as routing strategies. In our numerical evaluation, we showed that our approach is able to reach good accuracies. We illustrated that specific properties of network protocols such as resilience to packet loss can be explicitly included in the learned protocols by training the neural network with appropriate dropout.

Our work has inspired others to explore the network protocol design using GNNs, as shown by the work from Xiao *et al.* [191] and from Sawada *et al.* [147].

4.5 Graph models for reasoning about network protocols

> **Note** This section is based on [70] published in *IFIP Networking 2019*, 2019. The complete work is in Appendix A.2.3.

Automated approaches can greatly improve the trustworthiness of networks and hence reliability, by allowing to test a large number of network configuration for their policy compliance. Yet, many network verification tools still require a super-polynomial runtime to test basic connectivity properties, such as for example the works from Kazemian *et al.* [100] with Header Space Analysis (HSA), Anderson *et al.* [6] with NetKAT, or from Jensen *et al.* [95] with P-Rex. Testing whether network configurations are policy compliant even under failures, introduces another combinatorial complexity.

Schmid and Srba [151] recently showed that for the widely deployed Multiprotocol Label Switching (MPLS) networks, a polynomial-time "what-if analysis" is possible: an automata-theoretic approach, leveraging a connection to prefix rewriting systems, can be used to test important properties such as connectivity (can two endpoints reach each other?), loop-freedom (may packets be forwarded in circles?) or waypoint enforcement (is traffic always going through the firewall?), even under failures.

While this is promising, the runtime in practice is still relatively high (in the order of an hour even for relatively small yet complex networks). The approach from Schmid and Srba requires the construction of a large pushdown automaton (PDA), based on the network configuration, the routing tables, as well as the query. The PDA is then solved using reachability analysis.

In [70] we proposed to ask the question whether it is possible to build upon these recent results while exploiting opportunities for speeding up verification as well as to support an automated fixing of configurations. This is challenging also because unlike other networks, MPLS supports arbitrary (and in principle unbounded) header sizes: additional labels are for example pushed to route around railed links.

Our work is motivated by the goal to predict and fix properties according to a natural reg-

ular query language [95]. A (reachability) query is of the form

$$< a > \quad b \quad < c > \quad k$$

where the regular expression a describes the (potentially infinite) set of allowed initial label-stack headers, the regular expression b describes the set of allowed routing traces through the network, and the regular expression c describes the set of label-stack headers at the end of the trace. Finally, k is a number specifying the maximum allowed number of failed links.

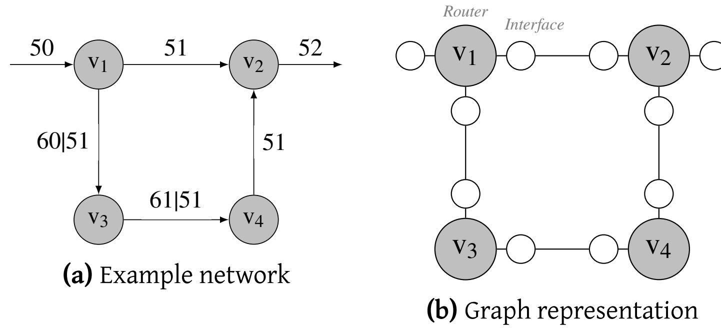
In our work, we contribute a novel approach to speed up verification and synthesis of the policy-compliance of network configurations. At the heart of our our tool, DeepMPLS, lies a new application of GNNs. Leveraging deep learning, DeepMPLS allows to predict counter examples (i.e., "proofs" or witness traces) to specific network properties (or queries), which can be verified fast. In fact, we show that DeepMPLS's probabilistic approach may even be used for *synthesis*: it has the potential to predict which MPLS rules should be added, in order to *re-establish* certain properties. The tool may hence overcome the need to perform more rigorous and time-consuming analyses in many scenarios.

Graph transformation

Our approach follows the idea introduced in Section 2.4 and the lessons learned in Sections 3.4 and 4.3. We model MPLS networks and the regular query language from P-Rex [95] as graph, and via the use of GNNs, predict various properties about the network.

The transformation process we propose in this paper is illustrated in Figures 4.8 to 4.10, where the MPLS network depicted in Figure 4.8a is transformed into a graph.

We first map the physical network topology to graph nodes as illustrated in Figure 4.8. Each router $v \in V$ in the network is represented as a node. Each network interface $i \in I_v^{in} \cup I_v^{out}$ is also represented as a node, connected via an edge to its router. Links are represented as edges connecting the two corresponding network interfaces.



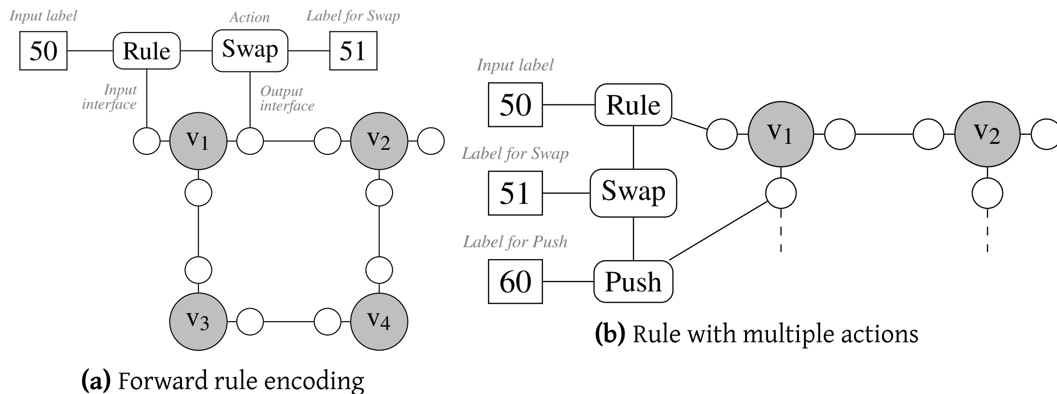
(a) Example network

(b) Graph representation

Fig. 4.8: DeepMPLS graph transformation

As presented in Figures 4.9a and 4.9b, the MPLS configuration of each router is also encoded as nodes and edges in the graph. Each MPLS label $l \in L$ is represented as a node. The routing table of each router $\tau_v : I_v \times L \rightarrow (2^{I_v \times Op^*})^*$ is represented as a set of rules. Each rule is represented as a node in the graph, connected the nodes representing its input interface $i \in I$ as well as its input label $l \in L$. The actions $o \in Op$ associated to a rule are also represented as nodes, connected via edges in case multiple actions are to be performed for a given rule as

illustrated in Figure 4.9b. MPLS actions with label parameters such as *swap* or *push* are connected to their respective label node. The last action associated to a rule is connected to its output interface.

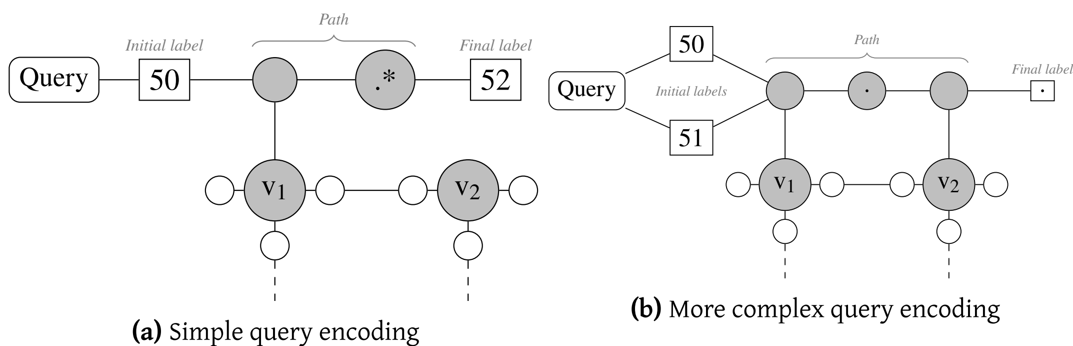


(a) Forward rule encoding

(b) Rule with multiple actions

Fig. 4.9: DeepMPLS graph transformation of MPLS forwarding rules

Queries are also encoded as nodes in the graph as illustrated in Figure 4.10. We follow an approach inspired by the McNaughton-Yamada-Thompson algorithm [128] which transforms a regular expression into an equivalent nondeterministic finite automaton. The different symbols of the regular expression of a query are represented as nodes, with edges representing their relationships. In case a symbol corresponds to a MPLS label or a router in the network, we reuse the node which was already defined in the graph. Wildcard symbols are represented as special nodes in the graph as illustrated in Figure 4.10a. Relationship between symbols such as combinations (*e.g.* $a_1 + a_2$) are represented using edges in the query representation, as illustrated in Figure 4.10b.



(a) Simple query encoding

(b) More complex query encoding

Fig. 4.10: DeepMPLS graph transformation of MPLS forwarding rules

Each node in the graph may have input features describing characteristics of the node. In our case, nodes are mainly represented by their type, encoded as categorical value. We define the 12 following types for the nodes:

- Elements of the network topology: *Router*, *Interface*;
- Elements of the MPLS configuration: *Rule*, *Label*, *Push Action*, *Swap Action*, *Pop Action*;

– Elements of the query and the regular expression: *Query*, *Label Wildcard*, *Label Dot*, *Router Dot*, *Router Wildcard*.

Depending on the task to perform, a classification task is then used on specific nodes or edges in the graph.

Numerical evaluation

We evaluate here the training of DeepMPLS for the *Satisfiability* task, namely prediction of the satisfiability of a query given an topology and MPLS configuration. Figure 4.11 illustrates the accuracy of DeepMPLS during training according to the number of training iterations, on both the training and the test dataset. Each training iteration corresponds to 16 analyzed topologies and queries from the training dataset, i.e. their representations as graphs. After 2500 training iterations, DeepMPLS reaches the accuracy of the baseline on the test dataset, before converging after around 25 000 training iterations.

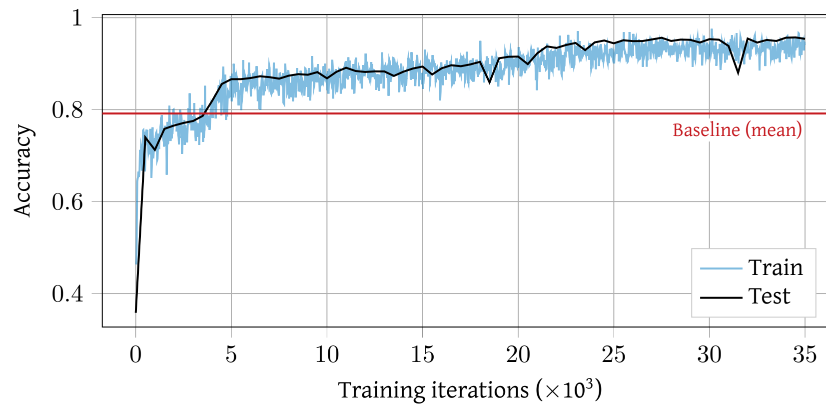


Fig. 4.11: Training of DeepMPLS for prediction of query satisfiability and comparison against baseline

In order to understand the practical applicability of DeepMPLS, we evaluate in this section its execution time in different settings. Since the neural network can be evaluated on either Central Processing Unit (CPU) or Graphic Processing Unit (GPU), we evaluated DeepMPLS on both platforms. Figure 4.12 illustrates the different execution times and compares DeepMPLS against P-Rex [95].

For the three different evaluations, we note a linear relationship between size of the push-down automaton – and hence size of the analyzed graph – and the execution time. DeepMPLS is one order of magnitude faster than P-Rex when running on CPU, and two order of magnitudes faster on GPU, mainly due to the better ability of GPUs of parallelizing the numerical operations used in neural networks. Those figures illustrate that DeepMPLS show promising applicability due to fast computation times.

Conclusion

In [70] and this section, we showed that deep learning can not only be used for faster verification of the policy-compliance of MPLS configurations, but even has the potential to provide

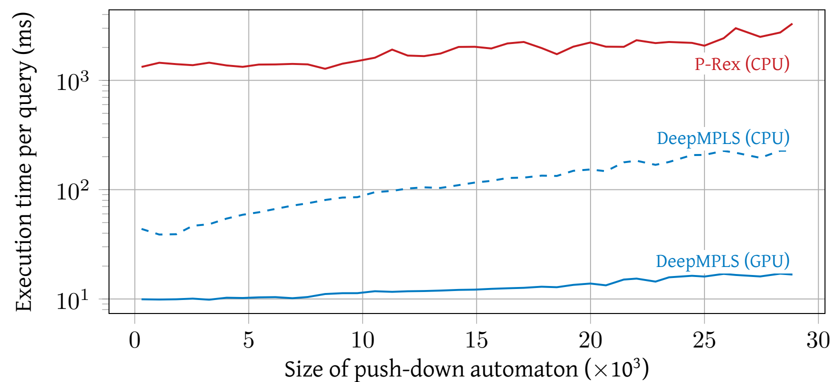


Fig. 4.12: Execution time of DeepMPLS on CPU and GPU compared against P-Rex for the Satisfiability task

efficient synthesis, automatically re-establishing certain network properties.

In general, we understand our work as a first step and believe that our work opens several interesting directions for future work. In particular, we believe that our approach can be refined and optimized further, to provide an even better performance. Furthermore, it will be interesting to investigate the synthesis of full MPLS configurations

Our work has inspired others to explore the verification of protocol and configuration using GNNs, as shown by the work from Bahnasy *et al.* [10] which looked at verification of Border Gateway Protocol (BGP) configurations using a similar approach than the one presented here.

4.6 Conclusion on machine learning methods for computer networks

We contributed in this chapter applications of GNNs and the concept introduced in Section 2.4 to various problems in networking: performance evaluation, routing and configuration verification and partial synthesis.

Additionally to our contributions in this chapter and in Chapter 3 and our review of GNNs in networking in Section 4.2, we showed that GNNs are a powerful and efficient method for solving some challenges in networking outside of network calculus (NC). This is especially relevant in challenges requiring a global view of the network topology.

5. HARDWARE AND SOFTWARE FOR EFFICIENT PACKET PROCESSING

5.1 Introduction **51**

5.2 Related work **52**

5.2.1 Advanced programmable dataplanes 52

5.2.2 In-network functions 52

5.2.3 In-network machine learning 53

5.3 Modular packet processing for fast prototyping **53**

5.4 Advanced secure hashes for in-network packet processing **56**

5.5 Adaptive ML-based batching for fast software routers **59**

5.6 Conclusion on efficient packet processing **62**

5.1 Introduction

We contributed in Chapters 3 and 4 various analytical methods for doing performance evaluation or inference about network properties using data-driven methods. One outcome for using those tools is offline *what-if* analysis, where assumptions and behavior of networks can easily be checked by network engineers. Another outcome would be to introduce those functions into the network itself and perform online changes, meaning having parts of the network dynamically reacting to changes (*e.g.* link failure, as illustrated in Section 4.4).

For these online evaluation, some functionalities would need to be directly performed inside the network (*e.g.* accurate packet monitoring), with what is known as *in-network computing*. In-network computing has demonstrated superior performance (*e.g.* NetCache from Jin *et al.* [96], or Netchain Jin *et al.* [97]) and it is also power efficient (*e.g.* work from Tokusashi *et al.* [178]). As contributed in Section 5.5 and also reviewed in Section 5.2.3, it is even possible with today's hardware to perform machine learning (ML) inference in the dataplane.

In this chapter we illustrate our contributions on P4, one of most recent solution for advanced dataplanes. We review how it may be used in an industrial environment, and also illustrate how advanced data-structures and security functionalities can be added to these dataplanes. We also evaluate an application of ML for automatically adapting the batching strategy of Vector Packet Processing (VPP) based on random forests.

5.2 Related work

We review in this section various related work on advanced dataplane, their technologies, and their applications.

5.2.1 Advanced programmable dataplanes

Approaches towards a top-down description of data-plane in a high-level programming language have been proposed since the late 1990s and early 2000nd. Kohler *et al.* [105] proposed Click, one of the early solutions which enables flexible packet processing in software, but with the drawback of difficulty regarding compilation to dedicated hardware.

More recently with the increasing use of Field Programmable Gate Arrays (FPGAs) for packet processing, Brebner and Jiang [36] proposed the PX programming language with a compiler targeting FPGAs. Dedicated hardware for packet processing such as Network Processing Units (NPUs) [76] or Reconfigurable Match Table (RMT) [26] have also been proposed. Song [165] proposed Protocol-Oblivious Forwarding (POF), which defines an Flow Instruction Set which is used for processing packets. Part of these work lead to the advent of P4 by Bosshart *et al.* [27] – already discussed in Section 2.3 – which attracted a lot of work for advanced dataplanes.

Regarding purely software-based packet processing on commodity multi-core processors, various works have been performed on the performance of such platforms. Dobrescu *et al.* [49] evaluated the predictability of Central Processing Unit (CPU)-based platform. They evaluated how contention for shared hardware resources such as caches can be taken into account for improving performance predictability, an important aspect in case of safety critical applications. Emmerich *et al.* [53] benchmarked various Linux-based software stacks for software-based packet processing and identified various bottlenecks responsible for poor performance.

Finally, Broadcom proposed the Software Development Kit Logical Table (SDKLT) and open sourced the Network Programming Language (NPL) specification [37] in 2019, currently targeting Broadcom Application-Specific Integrated Circuits (ASICs). NPL addresses similar goals than P4, with a similar match+action tables architectures. NPL is accompanied by a compiler suite and programs can be targeted to different architectures.

5.2.2 In-network functions

Based on the advances in dataplane programmability from P4, various works were done on adding functions in the network. Dang *et al.* [46, 47] brought the Paxos consensus protocol [110] to P4. Katta *et al.* [99] and Miao *et al.* [130] both used P4 for performing in-network load balancing. Tokusashi *et al.* [177] investigated key-value stores in P4 based on a FPGA implementation.

Various works focused on advanced network statistics. Huang *et al.* [87] introduced Sketch-Learn, a novel sketch-based measurement framework that resolves resource conflicts by learning their statistical properties to eliminate conflicting traffic components. Yang *et al.* [195] also proposed a similar sketch-based mechanism to perform statistical analysis on traffic. Ben-Basat *et al.* [14] proposed PRECISION, an algorithm that uses Probabilistic Recirculation to find top flows on a programmable switch. Chen *et al.* [41] proposed ConQuest, a compact data structure that identifies the flows making a significant contribution to the queue.

Finally, various proposals were made for advanced packet scheduling using P4. Sivaraman *et al.* [164] introduced the push-in first-out queue (PIFO) data-structure, and showed that it can be used for a wide variety of scheduling algorithms. More recently, Sharma *et al.* [159] proposed a way to achieve fair-queuing using P4.

5.2.3 In-network machine learning

Additionally to the advanced function previously reviewed, some recent works also focused on performing in-network ML inference. The first steps toward in-network inference were works from Siracusano and Bifulco [162] and from Sanvito *et al.* [146], which discussed the implementation of binary neural networks (BNNs) within programmable network devices using mostly P4 primitives.

Subsequent works focused on using additional hardware or P4 externals to provide in-network inference. Xiong and Zilberman [193] implemented traditional ML algorithms such as decision tree, Support Vector Machine (SVM), naïve Bayes and K-means in P4, and performed an evaluation on a FPGA as-well-as on P4's reference software implementation. Li *et al.* [118] proposed an implementation of reinforced learning within a switch, but used a bespoke acceleration module. Neural networks (NNs) implementations were also proposed for NPUs, such as the works from Langlet [111] and from Siracusano *et al.* [163].

All these works illustrate that in-network ML inference is feasible, which could lead to interesting new applications of ML for network functionalities.

5.3 Modular packet processing for fast prototyping

Note This section is based on [73] published in *Proceedings of the 6th International Workshop on Aircraft System Technologies*, 2017, [71] published in *Proceedings of the 9th European Congress Embedded Real Time Software and Systems*, 2018. This work was also presented in [155] in *5th P4 Workshop*, 2018. The complete works are in Appendices A.3.2 to A.3.4.

We look in this section and in [73, 71, 155] at recent developments in advanced programmable dataplanes from the perspective of the avionic industry, with a focus on Avionics Full DupleX Switched Ethernet (AFDX) as a case-study.

Our main contribution is an analysis of those new solutions in the scope of aeronautical applications in terms of offered features and performance. We first investigate their applicability from a functional point of view and identify missing features of the current approaches. In a second step, we do a performance analysis using measurements on three different target hardware and investigate if those new developments and platform are sufficient for aeronautical applications from a performance point of view.

P4 as a fast prototyping platform

Distributed embedded electronic applications have become the norm in a large part of the aeronautical industry. Due to hard real-time and strict safety constraints associated with aircraft, specific equipment are generally designed and built in order to fulfill those constraints. When

such equipment are not available off-the-shelves, costly and time consuming developments have to be undertaken. A prevailing solution commonly used to address this issue is to employ FPGAs since they offer high customizability with high performance at moderate costs. A main drawback of FPGAs is that they require a high level of expertise and long development times to produce efficient and bug-free devices.

P4 promises various properties (listed earlier in Section 2.3) which make it attractive for the aeronautical industry. The main advantage of P4 is the decorrelation between the behavior of a packet processing device and the hardware which is used. It means that engineers are not tied to a specific set of network protocols implemented by hardware vendors, and that protocol development can be decorrelated from hardware development.

A second advantage of P4 is the simplicity and constraints put on the abstract forwarding model. Since P4 forbids dynamic memory allocation and iterations with unknown counts – unlike more generic programming languages such as C – formal derivations of worst-case execution time and resource usage of a P4 program are fairly straightforward. This means that per-packet latency, memory footprint and maximum throughput of a packet-processing pipeline can be determined at compile time, a feature highly desirable in the avionic industry for certification.

Finally, regarding the features supported by P4 in terms of packet processing actions, it covers most of the use-cases relevant for network protocols used by the aeronautical industry. Some specific mechanisms for Quality-of-Service (QoS) are missing, such as packet scheduling, but some proposals have been made to overcome this issue, such as the work from Sivaraman *et al.* [164].

Additionally, we also investigated an implementation of AFDX in [71], and showed that a simplified AFDX switch can be implemented using P4..

Numerical evaluation

Additionally to our assessment from a functional perspective, we also evaluate performance aspects of P4 in [73, 71]. The three following platforms were used for our evaluation:

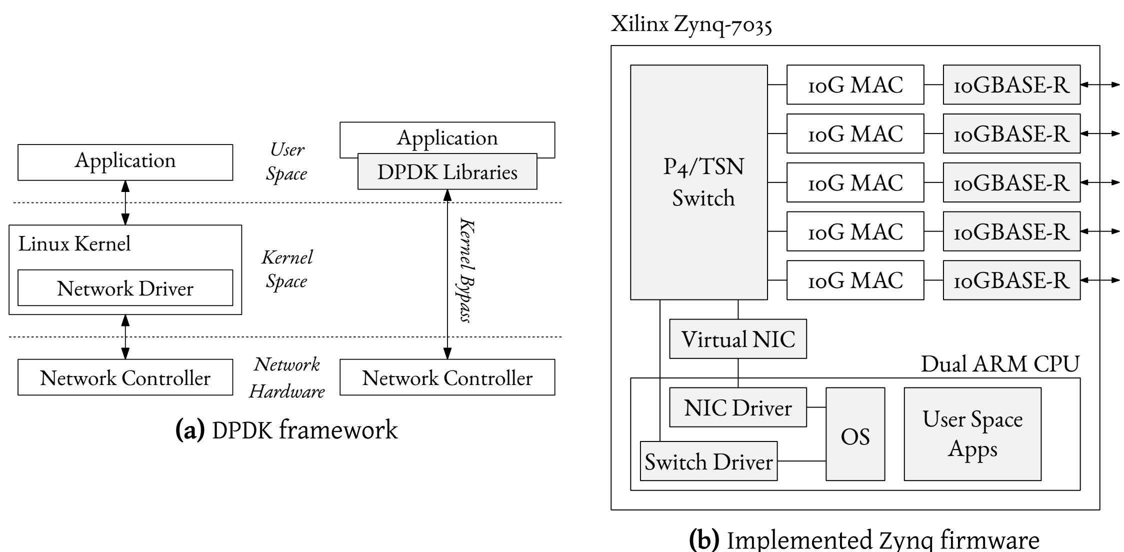
CPU For this platform, we used the Translator for P4 Switches (T4P4S) [106, 183] platform, a P4 compiler which generates platform-independent C code which can be linked with additional libraries, in our case Intel's Dataplane Development Kit (DPDK). This platform is illustrated in Figure 5.1a, where the kernel bypass is illustrated, enabling us to perform faster packet processing.

NPU A 10 Gbit/s Netronome Flow Processor (NFP)-4000 Agilio SmartNIC [89] is used for this platform. The NFP-4000 is a NPU that relies on a 32 bit many-core architecture with up to 60 freely programmable flow processing cores. For this platform, our evaluation is limited to the case where packet processing is still performed on CPU via the DPDK framework, meaning that additional delay is required to transfer packets between the NPU and CPU. As a benchmark we also implemented a simpler packet forwarder on the NPU which bypasses the CPU.

FPGA The FPGA target is based on the Xilinx Zynq-7035 multiprocessor system on a chip (MP-SoC). As illustrated in Figure 5.1b, the main part of the firmware is an Ethernet switch

with optional Time-Sensitive Networking (TSN) features. The switch is connected to the external 10GBASE-R ports, as well as to internal virtual network interface cards connected to the CPU. The packet processing and forwarding is performed completely by the switch core within the programming logic without any involvement of the internal CPU, allowing maximum throughput and minimum latency. For this evaluation, we wrote our own simplified P4 to VHSIC Hardware Description Language (VHDL) compiler.

We note that since the time of our evaluation, various commercial and open-source solutions have been proposed for compiling P4 to FPGA platforms (such as P4→FPGA [88] used later in Section 5.4).



(a) DPDK framework

(b) Implemented Zynq firmware

Fig. 5.1: Overview of the different platforms used for benchmarking

Figure 5.2 presents the packet processing latency as a function of the time between two frames (or framegap). We notice that for framegaps larger than 1 µs the processing latency is of 24 µs for packet sizes of 1518 B for the software-based and network processor platforms. Since both approaches require copies of the frames from the network cards to the CPU, similar latencies are expected. For framegaps smaller than 1 µs the processing latency of the software-based platform increases up to 1 ms depending on the packet size. The packet processing is not able to keep up with the incoming rate and packets are buffered leading to the increased latency.

We note that while the software platform does not provide us deterministic guarantees required for safety applications, such a platform might be interesting for functional testing or in services with short life-cycles and no hard real-time guarantees needed. Typical services such as passenger connectivity could fit these requirements, since on-board passenger devices have a fast update rate, with changing needs and protocols.

The network processor platform without CPU is able to better cope with the more intensive traffic, which can be explained by the fact that the processing is completely performed by the NPU, without CPU involvement or the need to copy packets.

The FPGA-based platform produces the best latencies, with values around 1.2 µs without buffering (i.e. for large framegaps). Once the internal bandwidth limit is hit, packets are

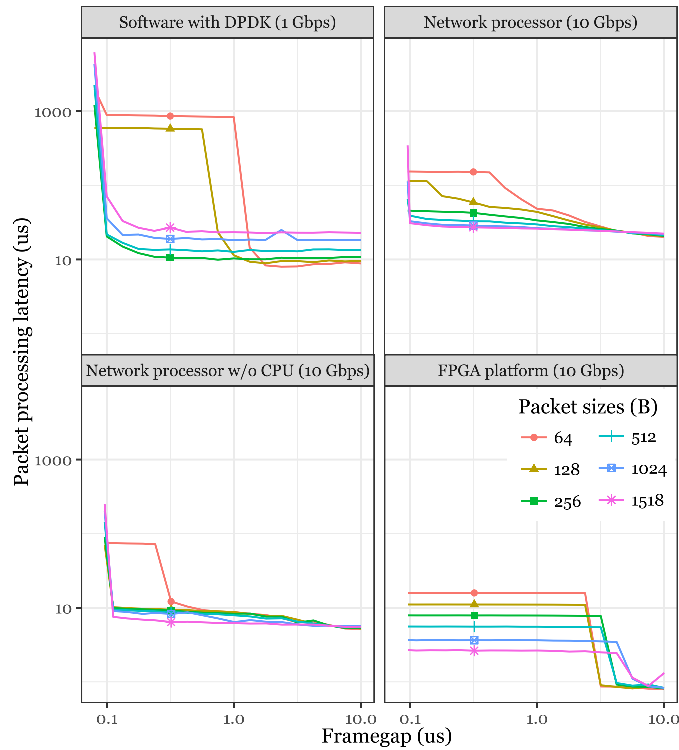


Fig. 5.2: Packet processing latency as a function of the time between two frames

buffered at ingress and eventually dropped. The latencies up to 15.8 µs observed in this region of small framegaps correspond to the capacity of the ingress buffers, which are much smaller compared to the software/network processor implementations and thus leading to smaller latencies in these situations, but potentially more packet losses during short, intense traffic bursts.

Conclusion

We showed here and in [73, 71] that P4's high flexibility in combination with simple building block enables a formal analysis make it an attractive platform for some network protocols used in aeronautical use-cases. While some features such as definition of advanced egress packet scheduling and methods for time-based or time-triggered protocols are still lacking for more advanced network protocols, recent additions to the language in $P4_{16}$ and the Portable Switch Architecture (PSA) make it an attractive platform. A case study of implementing AFDX was also performed in order to demonstrate that aeronautical protocols may be implemented using P4.

5.4 Advanced secure hashes for in-network packet processing

Note This section is based on [157] published in *Proceedings of the 2nd P4 Workshop in Europe*, 2019. The complete work is in Appendix A.3.1.

We show in this section and in [157] that advanced packet processing functions can be efficiently implemented in network switches and router. To enable authentication and resilience, we make the case for extending P4 targets with cryptographic hash functions. Hash-based data structures like hash tables, bloom filters, or count–min sketches often serve as a basis for efficiently tracking flows.

We propose an extension of the P4 PSA for cryptographic hashes and discuss our prototype implementations for three different P4 target platforms: CPU, NPU, and FPGA. While P4 does not directly offer primitives for working with data structures such as hash tables or Bloom filters, P4 primitives can be used in combination with our contributions via P4 externs and P4 registers to emulate those data structures. Each platform has its own way how P4 externs can be added.

For our evaluation, we focused on two types of hash functions. First, we evaluated the Secure Hash Algorithm (SHA) family of hash functions, which are strong candidates regarding cryptographic features and security. Due to the use of relatively small messages in packet processing, the choice of a hash function for efficient processing is not straightforward. In this case, the SHA-based hash functions are not well suited since were not designed with good performance for small inputs. Hence, we also focused on the SipHash family of hash functions from Aumasson and Bernstein [8] used in various programming languages and software, a hash function designed for good performance for small inputs.

CPU We extended here the T4P4S platform, already used in Section 5.3. As T4P4S only supports the TCP/IP checksum calculation as hash algorithm, we extended it with open-source implementations of SipHash and HMAC-SHA. Our measurements were performed on a server equipped with an Intel Xeon CPU E5-2620 v3 (Broadwell) at 2.40 GHz with an Intel X540 network card supporting 10 Gbit/s Ethernet.

NPU As in Section 5.3, Netronome's NFP-4000 Agilio SmartNIC is used here. Compared to Section 5.3, we bypass the CPU to perform the P4 packet processing only on the NPU via Netronome's P4 compiler. The SmartNIC allows implementing P4 externs in Micro-C, a variation of C used to program the processing cores. Externs are inlined into the compiled P4 program. We implemented the SipHash-2-4 function in Micro-C, calculating a hash for the payload of the Ethernet frame.

FPGA Our work on the FPGA platform is based around the NetFPGA SUME board from Zilberman *et al.* [201] and the P4→FPGA from Ibanez *et al.* [88]. Open-source SipHash and SHA3 IP cores were integrated into our prototype design. Our initial evaluation showe that a seamless integration as P4 externs via interfaces defined by the P4→FPGA implementation was not possible. Hence, we decided to change the P4 switch model of the P4→FPGA design to integrate the hash calculation in the egress path after the synthesized P4 program as illustrated in Figure 5.3.

Numerical evaluation

For our numerical evaluations, our measurement setup consists of two servers connected via a 10 Gbit/s Ethernet link. One server acts as a load generator and sends packets to the device

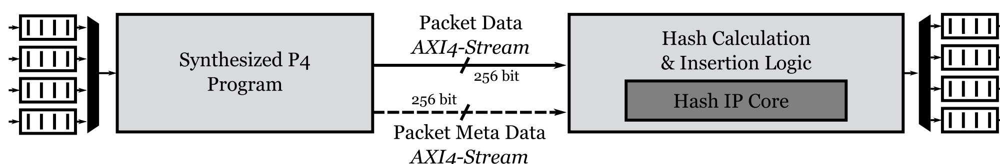


Fig. 5.3: Integration of hash calculation and insertion for the FPGA platform

under test, which runs a layer 2 forwarding P4 program that additionally calculates hashes based on the complete Ethernet frames.

Results for maximum throughput are presented in Figure 5.4. Independent of packet size, all three platforms reach 10 Gbit/s in the baseline scenario, with the exception for minimum-sized packets on the CPU target only achieving 95.03 %. Adding the calculation of hashes reduces the maximum performance such that no platform can reach line rate for packets with minimum size. In our evaluation, the best results are achieved by the NPU for packet smaller than 900 B. For larger packets, slower shared Random Access Memory (RAM) has to be accessed, causing a sharp drop in throughput. For the FPGA, while our prototype is limited to open-source hash implementations, we note that higher throughput could be achieved with commercial IP cores.

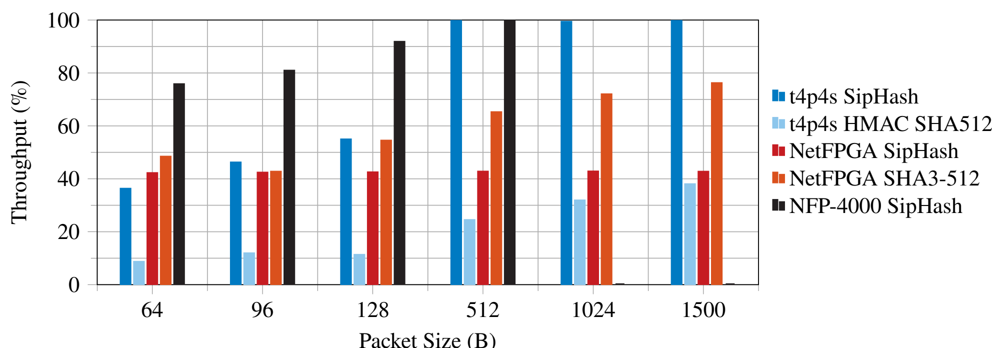


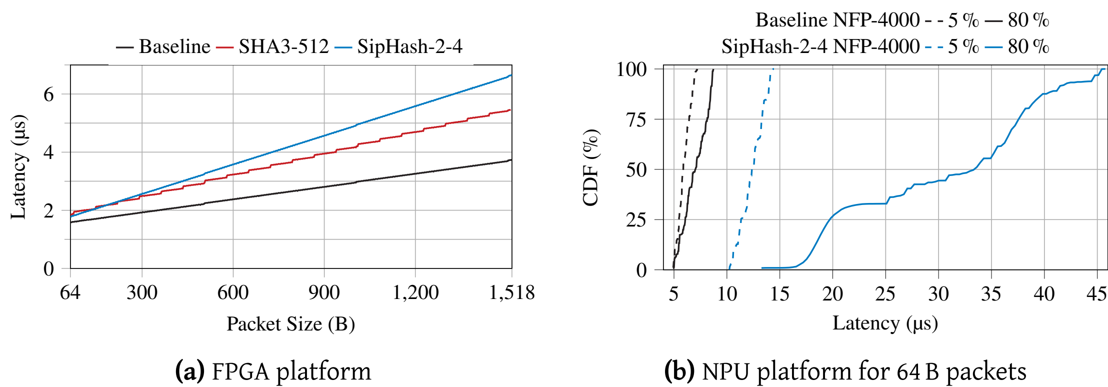
Fig. 5.4: Throughput achieved on the different platforms

Figure 5.5a shows our latency measurements for the NetFPGA SUME. As expected, latency increases linearly with packet size with slight discontinuities due to the block-based hash calculation. We found that for each packet size the measured values do not differ by more than 100 ns.

The NPU demonstrates stable behavior below 10 µs with no outliers for the baseline scenario, as illustrated in Figure 5.5b. Performing the SipHash-2-4 operation shifts the latency distribution to the right up to 30 µs and increases the long tail. For the CPU platform, overall the latency is between 10 µs and 80 µs, however, outliers, which regularly occur when using DPDK.

Conclusion

Our works revealed two insights about the current use of hash functions in P4 applications. First, a prevalent use of Cyclic Redundancy Check (CRC), making applications vulnerable to



(a) FPGA platform

(b) NPU platform for 64 B packets

Fig. 5.5: Latency achieved on the different platforms

potential attacks targeting hash collisions. Second, protocols and applications requiring cryptographic hashes for authentication or integrity cannot be described using P4. Therefore, the implementation of cryptographic hash functions would increase the applicability of P4 to a wider range of use cases.

Our measurements show hashing performance to be highly target, algorithm, and use case specific. Therefore, we cannot recommend a one-size-fits-all solution. We rather suggest that P4 targets should implement hash functions – operating on header and payload data – from a family of algorithms, which should be recommended by the P4 specification. These recommendations should include cryptographic hashes and take into account the unique characteristics of platforms such as CPU, NPU, FPGA, or even future dedicated ASICs.

5.5 Adaptive ML-based batching for fast software routers

> Note This section is based on [139] published in *Proceedings of the 7th IEEE International Conference on Network Softwarization*, 2021. The complete work is in Appendix A.3.5.

We show in this section and in [139] that ML inference can be efficiently implemented in software routers for dynamically adapting its behavior.

Modern tools for software packet processing, incorporate many optimizations such as processing packets in batches and adopting a kernel-bypass approach to access the Network Interface Cards (NICs) with pure user-space drivers and minimize the interference of low-level system calls by the operating system. This strategy is implemented in VPP for example, a high speed packet processor originally developed by Cisco and recently released as an open-source [142].

In particular, batching is usually adopted in conjunction with a busy polling behavior: the CPU continuously performs a loop to verify if any packet is received at the NIC, then it uses a minimalist batch creation algorithm to process a full batch of packets (as opposed to per-packet processing) and it repeats the loop at the end of the processing. Batching and busy polling are very effective in high-load scenarios, where the cost of interrupt handling per packet could lead to a saturation of the CPU.

While the maximum batch size is fixed (usually between 32 and 512), the actual size of the batches depends on the number of packets waiting in the NIC's input queues. On the other hand, the actual batch size also affects the efficiency of the processing, with small batches requiring more clock cycles per packet compared to large batches. This causes a feedback loop, where oscillating batch size can be observed in scenarios where the input load does not fully saturate the CPU.

We contribute in [139] an ML-based solution to dynamically allocate batch sizes depending on the traffic condition instead of the classical busy polling approaches. It was already shown (*e.g.* by Miao *et al.* [129]) that such adaptive batching methodology is beneficial to software routers.

Our architecture is illustrated in Figure 5.6. Our modified software router consists of two parallel components: the software router, and the ML process. Every time the `dpdk-input` node submits a batch of packets to the processing graph, it also communicates the batch size to the ML process. The ML algorithm then runs its predictions and returns the new, updated action instructions which are in turn read by VPP. For the communication between the two processes we adopt non-blocking I/O in order to keep high throughput performance.

The ML process uses a list of the last batch sizes as input for its actions. As output, the ML process infers a sleep duration, which will temporarily pause the VPP process. This frees the CPU from busy polling the NIC queues, hence saves the CPU for other resources.

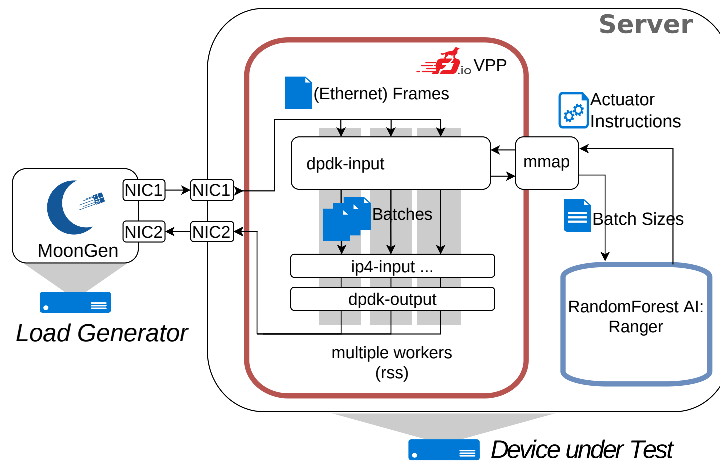


Fig. 5.6: Testbed used for our experimental evaluation, which includes the traffic generator, the device under test and the AI components

Numerical evaluation

We numerically evaluate in here the different components of our architecture. We also illustrate the impact that our machine-learning based software router has on the overall performance. As presented in Figure 5.6, our testbed is composed of VPP and the ML process run on the device-under-test, while load scenarios are performed by the load generator running MoonGen from Emmerich *et al.* [52]. The traffic generated by MoonGen consists of 64 B packets sent at different constant rates.

Stage of Integration	Throughput	Ratio
Unmodified VPP	14.15 Mpps	100 %
Logging only	13.95 Mpps	99 %
Logging + Exporting	13.94 Mpps	99 %
...+ Exporting + Ranger load	11.57 Mpps	82 %
...+ Final trained forest	12.26 Mpps	87 %

Tab. 5.1: Maximum throughput at different stages integration.

Table 5.1 summarizes the impact of data collection and processing on VPP's throughput. Running VPP while logging all batch sizes into the shared memory, results in a maximum throughput of 99 %. Running ranger for a single prediction, meaning exporting the ring buffer only once to ranger, results in a similar throughput performance of 99 %. When running the ring buffer export and prediction in an endless loop, VPP's throughput drops to 11.57 Mpps which corresponds to 82 % of the performance of unmodified VPP.

Finally, we evaluate our architecture by measuring the CPU utilization in different scenarios and comparing it to an unmodified VPP. Results are presented in Figure 5.7. Our ML process updating the optimization instructions on CPU2 constantly utilizes about 98 %. The worker thread of VPP on CPU1 shows a different behavior when using our approach. When offered no load, the sleep time is set to 30 by the forest. This results in a utilization of only around 20 % on CPU1. When offered more load (1000 Mbit/s around time 20 s in Figure 5.7), the sleep time drops and CPU1 raises to 45 % load to process the packets. From offering 1200 Mbit/s of load onward (at time 31 s in Figure 5.7), VPP's worker thread starts hitting the upper limit of available cycles of CPU1. At a throughput of 12.26 Mpps the maximum performance of the system is finally found.

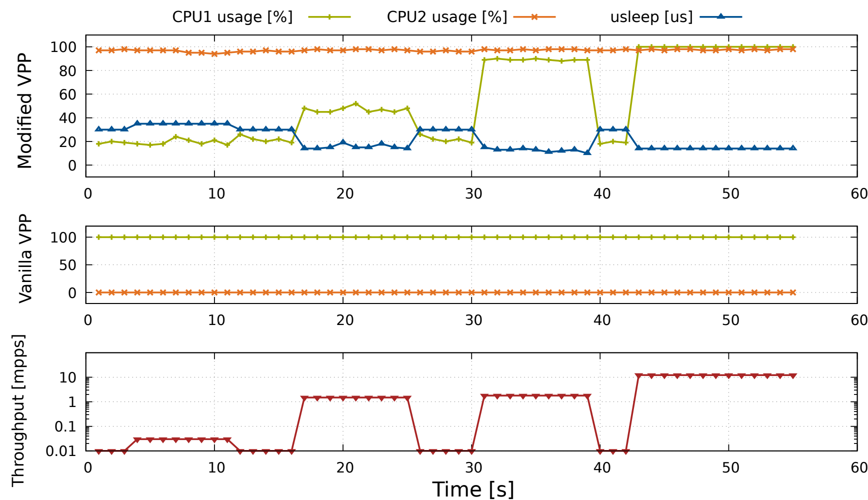


Fig. 5.7: A comparison under the same load scenario. Unmodified: CPU1: VPP worker, CPU2: nothing. Modified: CPU1: VPP worker, CPU2: ranger.

Conclusion

We contributed in this section an approach for including ML inference in the dataplane. We showed that the CPU utilization of VPP could be reduced in low load scenarios using random forests for finding optimization parameters at runtime. Regardless of the added complexity, the throughput performance in high load situations is reduced by only 13 %. Although a separate core is fully utilized by the ranger thread, we illustrate that the same core can be used to save CPU time off VPP worker threads on multiple cores. Our work illustrate that more advanced packet processing is possible with today's advanced hardware and software.

5.6 Conclusion on efficient packet processing

We contributed in this chapter a functional and performance evaluation of P4, a P4 extension for facilitating advanced data-structures and security functionalities, as well as an extension of VPP to include a ML-based adaptive batching mechanism.

We showed that P4 is flexible and can be used in an industrial environment with industry-specific network protocols. With our performance evaluations, we showed that advanced packet processing – either based on software solutions such as DPDK, or hardware-accelerated solutions such as NPUs or FPGAs – can come with a 10 Gbit/s on commodity hardware. P4 is also flexible enough to enable additional functionalities, as reviewed in Section 5.2 and evaluated in Section 5.4. In Section 5.5 we also showed that ML can be directly included in the dataplane to provide advanced optimization.

Overall, our contributions and related work illustrate that more advanced packet processing is possible with today's advanced software and hardware, even including ML inference in the dataplane. This current state of networking is encouraging for the adoption of more advanced packet processing based on in-network processing and accurate per-packet performance measurement. A good example of this is P4's In-Band Network Telemetry (INT) [172], which provides a standardized way of doing fine measurements and reporting them in standardized way [173]. Such features has even lead to advanced analytic software, such as Intel's Deep Insight Network Analytics Software [90], which enables real-time per-packet visibility with advanced monitoring and analysis of packet drops.

6. CONCLUSION

6.1 Summary and conclusions

In this Habilitation thesis we have studied and contributed data-driven methods for networking challenges. We presented both theoretical results based on mathematical frameworks, as well as more practical results based on real implementations and measurements on testbeds.

The main contribution of this research work was a data-driven approach based on graph transformation and graph neural networks (GNNs). This approach, introduced in Section 2.4 and illustrated in Figure 6.1, stemmed from our finding that various challenges found in networking research can be modeled as graph, a key data structure for representing network topologies, flows, their properties such as configuration, and also their relationships. Using this natural way of representing data and by adding expert knowledge, we built on the recent works from the machine learning (ML) community on GNNs to make efficient and accurate predictions about the problem to be solved.

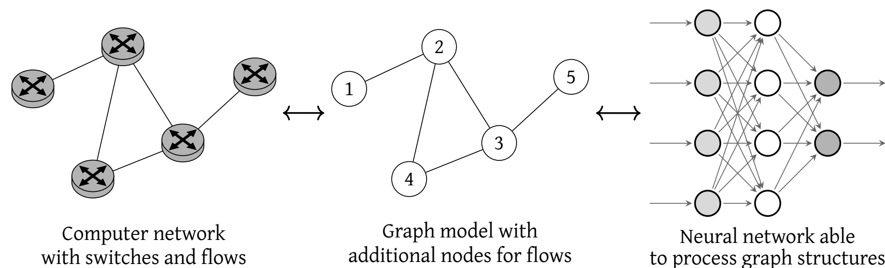


Fig. 6.1: Graph approach proposed in this Habilitation thesis

We demonstrated in Chapters 3 and 4 that this approach can be applied to a variety of networking challenges, from formal analysis of networks, to design of network protocols.

In the field of network calculus (NC), we contributed in Section 3.4 and [69, 62, 64, 65, 75] the first applications of ML and GNN to NC analysis. One finding of our research was to illustrate that getting the tightest bound from NC requires expert knowledge since various analyses may be applied, each performing at its best on specific networks, and each coming at a given computational cost. With our work, we alleviated this need for expert knowledge by using a data-driven approach. We showed with DeepTMA and DeepFP in [62, 75] that ML has its place inside NC, opening up the door to formally-verified tight bounds competitive with state of the methods, but at a low computational cost. Our contributions in this field is illustrated in Figure 6.2.

In Chapter 3, we also contributed two extensions of NC. We showed in Section 3.3.1 how to use NC with multicast flows, enabling the key properties of NC – namely Pay Burst Only Once

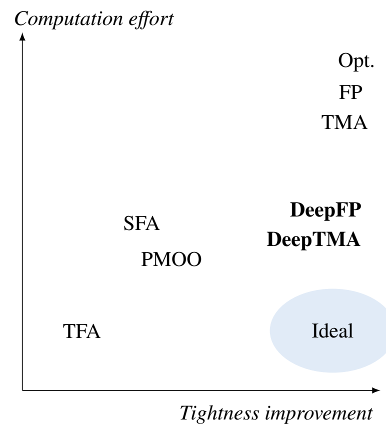


Fig. 6.2: Relationship between tightness and execution time of the different deterministic network calculus (DNC) analyses

(PBOO) and Pay Multiplexing Only Once (PMOO) – to be applied end-to-end on multicast flows. We also contributed in Section 3.3.2 the detouring concept, an approach which adds pessimism in the network in order to empower PMOO on longer tandems.

In Chapter 4 we demonstrated other applications of GNNs to networking challenges. With [60, 61] in Section 4.3, we contributed one of the first works applying GNNs to the performance evaluation of elastic and inelastic flows, predicting flow bandwidth and packet latency. We showed that GNNs can provide accurate and efficient predictions of flow performance, providing a fast tool for what-if analysis. With [68] in Section 4.4, we contributed a novel method taking advantage of the message passing principle of GNNs for building network protocols. We showed that routing protocols can easily be trained for, requiring no expert knowledge on network protocol design. Finally, with [70] in Section 4.5, we contributed an application of GNNs to the analysis of Multiprotocol Label Switching (MPLS) networks and configurations. We showed that our graph transformation is able to handle more advanced concepts such as MPLS forwarding rules and configuration.

Finally, in Chapter 5 we explored how lessons learned from data-driven approaches and their models may be included in the networks themselves. We focused on two packet processing platforms. First we work on P4, a promising solution for including such advanced functionalities in the dataplane on various hardware platforms such as Central Processing Units (CPUs), Network Processing Units (NPUs), Field Programmable Gate Arrays (FPGAs) or even dedicated Application-Specific Integrated Circuits (ASICs). We contributed in [73, 71] and Section 5.3 an evaluation of P4 in an industrial context, showing that P4 could be used for implementing an Avionics Full DupleX Switched Ethernet (AFDX)-like protocol. We also showed in [157] and Section 5.4 that P4 can easily be extended to include advanced functionalities such as secure and fast hash functions. Our performance evaluations showed promising results, achieving 10 Gbit/s line-rate in some cases. Secondly we work in Vector Packet Processing (VPP), a software router based on recent advances on software-based packet processing. We also contributed in [139] and Section 5.5 an approach for using ML inference in the dataplane for more efficient packet processing. We illustrate that by having a dynamic batching allocation, CPU time can be saved.

Overall, our contributions showed that a data-driven approach for networking is highly

relevant. By combining the correct data structure with the proper data processing algorithms – *i.e.* graphs and GNNs – we showed that some networking challenges can be solved at a low computational cost. As mentioned in the different conclusions of Chapters 3 and 4, our work inspired others to also apply GNNs and similar architectures to similar networking challenges (*i.e.* NC and performance evaluation) and also more diverse challenges (*e.g.* wireless networks or Border Gateway Protocol (BGP)). This shows that our approach has been validated by others and is also highly versatile.

6.2 Future research directions

Several open research questions are still open. We already mentioned throughout this Habilitation thesis and our works various improvements and possible extensions of the developed methods which could be addressed. We focus in the rest of this section on broader open research questions.

Improved graph transformations and GNNs models

We showed in our various works that the approach presented in Section 2.4 is efficient. Yet, we mainly focused on the same graph transformation and the same GNN architecture based around the gated graph neural network (GGNN) from Li *et al.* [119]. While working on improvements of DeepTMA and DeepFP, numerical evaluations showed that some graph nodes and their features were less important than others on the outcome of the prediction. We showed for example in [75] that some type of nodes could entirely be dropped without deeply affecting the accuracy of the GNN. This means that there is space for optimization with our various graph transformations, potentially making them lighter for faster processing, or more descriptive for better accuracy.

Similarly, since our first use of GGNNs in [60], various improvements in the ML community have been made on GNNs. While our early benchmarks showed only minimal gains of changing the GNN architecture, it is worthwhile to further investigate this. Other aspects of our ML pipeline could be improved, such as better dataset generation or better hyper parameter tuning.

Deeper integration between formal methods and GNNs

Following on the successes of DeepTMA and DeepFP, we believe that there is still some gains to be made in NC by integrating GNNs more deeply in the network analysis. With our works on Tandem Matching Analysis (TMA), Flow Prolongation (FP) or detouring, we illustrated that there are still various ways to achieve tightness in NC, each with their own way at achieving this goal. A unified solution taking advantage of these with GNNs predictions could be explored.

There are also some aspects of NC which we did not explore. We focused on blind-scheduling and First In First Out (FIFO) scheduling, but other mechanisms could be evaluated. Other types of flows – such as flows with circular dependencies – or more advanced arrival and service curves may also be addressed. Finally, the methods we developed may also be applied in other parts of NC, such as stochastic network calculus (SNC).

Network optimization and configuration synthesis

We illustrated in Section 3.4 that GNNs can be used to avoid computationally expensive exhaustive searches, coming up with the optimized solution at almost no computational cost. The same concepts may also be applied to more traditional network optimization problems, which are often NP hard due to their combinatorial aspect. Examples include flow routing, virtual network function (VNF) placement, or network design.

We also explored in Section 4.5 how to apply GNNs to network configuration. Manual network configuration is still a common practice nowadays, leading sometimes to misconfigurations and network failures. Automation of formally correct synthesis of configurations would be an essential tool, where GNNs may help speeding up part of the process.

Automatic network protocol design

We contributed in Section 4.4 an approach for applying GNNs to the task of routing network protocols generation. While we focused on routing in wired networks, the same approach should also be applicable to routing in wireless networks. Due to the one-to-many communication scheme of wireless networks, some adjustments of the network representations and use of hypergraphs would be required.

Adjacent to that, other aspects of protocol design would be required for a deployment in a real network. Investigation about methods to achieve guarantees about the generated protocol would be needed. A promising approach to this challenge would be to investigate hybrid protocols, where parts of the new protocol would be designed by more traditional tools and network experts, while other parts would be generated by ML.

Fast and flexible packet processing in the dataplane

Novel approaches for packet processing have been proposed in the last years for offering more flexibility and performance to network engineers. These approaches follow the trend of bringing more software in the network, a trend first started with Software Defined Networking (SDN) for the control plane, now also being pushed for the data plane. As we illustrated in Chapter 5, these novel platforms are efficient and can process packets at line rate.

Since the publication of our investigations, various other hardware platforms were also proposed on the market (*e.g.* Intel's Tofino). Investigation of the performance and flexibility of these platforms would shed some light on the potential of these new technologies. Additionally, as recently showed by [58], work is still needed to remove limitations of these platforms towards their large scale deployment.

Application to other areas of networking

Finally, we focused in this Habilitation thesis on a few network challenges and research areas: deterministic network calculus (DNC), performance evaluation, MPLS configurations, and network routing. We believe that our approach is applicable to other areas of networking where graphs are the correct data structure to work with. As mentioned throughout this thesis, our work inspired other to also use GNNs, showing that it is a versatile tool.

APPENDIX

A. PUBLICATIONS

A.1 Tight and efficient bounds in network calculus with fast heuristics . . . 70

A.1.1 Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows 70

A.1.2 Deterministic Network Calculus Analysis of Multicast Flows 79

A.1.3 Virtual Cross-Flow Detouring in the Deterministic Network Calculus Analysis 95

A.1.4 The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds 101

A.1.5 DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks 108

A.1.6 On the Robustness of Deep Learning-predicted Contention Models for Network Calculus 118

A.1.7 Graph-based Deep Learning for Fast and Tight Network Calculus Analyses 126

A.1.8 Tightening Network Calculus Delay Bounds by Predicting Flow Prolongations in the FIFO Analysis 141

A.2 Application of graph-based deep learning methods for computer networks 156

A.2.1 Performance Evaluation of Network Topologies using Graph-Based Deep Learning 156

A.2.2 DeepComNet: Performance Evaluation of Network Topologies using Graph-Based Deep Learning 165

A.2.3 DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning 182

A.2.4 Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning 192

A.3 Hardware and software for efficient packet processing 199

A.3.1 Cryptographic Hashing in P4 Data Planes 199

A.3.2 Rapid Prototyping of Packet Processing Devices for Aeronautical Applications 206

A.3.3 Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications 217

A.3.4 Rapid Prototyping of Avionic Applications Using P4 227

A.3.5 Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning 229

A.4 List of publications 235

This appendix contains the complete version of all the publications mentioned in this habilitation thesis.

A.1 Tight and efficient bounds in network calculus with fast heuristics

A.1.1 Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows

This work was published in *Proceedings of the 10th International Conference on Performance Evaluation Methodologies and Tools*, 2016 [19].

Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows

Steffen Bondorf
Distributed Computer Systems (DISCO) Lab
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
bondorf@cs.uni-kl.de

Fabien Geyer
Airbus Group Innovations
Dept. TX2P
D-81663 Munich, Germany
fabien.geyer@airbus.com

ABSTRACT

Guaranteeing performance bounds of data flows is an essential part of network engineering and certification of networks with real-time constraints. A prevalent analytical method to derive guarantees for end-to-end delay and buffer size is Deterministic Network Calculus (DNC). Due to the DNC system model, one decisive restriction is that only unicast flows can be analyzed. Previous attempts to analyze networks with multicast flows circumvented this restriction instead of overcoming it. E.g., they replaced the system model with an overly-pessimistic one that consists of unicast flows only. Such approaches impair modeling accuracy and thus inevitably result in inaccurate performance bounds.

In this paper, we approach the problem of multicast flows differently. We start from existing DNC analyses and generalize them to handle multicast flows. We contribute a novel analysis procedure that leaves the network model unaltered, preserves its accuracy, allows for DNC principles such as pay multiplexing only once, and therefore derives more accurate performance bounds than existing approaches.

CCS Concepts

•**Networks** → **Network performance evaluation;** *Network performance analysis;* Network performance modeling; •**Computing methodologies** → **Symbolic and algebraic algorithms;** *Symbolic calculus algorithms;*

Keywords

Delay bounds, deterministic network calculus, feed-forward networks, multicast flows

1. INTRODUCTION

Distributed embedded electronic applications communicating via packet networks have become the norm in various industries such as automotive, avionic or automation. In such industrial applications, real-time constraints on packet delay and jitter are usually required in order to ensure the specified processes behavior. Due to certification of systems as well as reliability demands, formal methods are applied to validate these timing constraints. They allow for hard guarantees via upper bounds. While different analytical methods have been proposed in the literature, Deterministic Network Calculus (DNC) established itself as common tool to analyze asynchronous communications in packet networks. A concrete example of this is Avionic Full-Duplex Ethernet (AFDX), a communication technology based on Ethernet and already deployed in avionic systems. Network calculus has proven a key tool for the certification of the deterministic property of the networks used for fly-by-wire [10].

An important property of those industrial networks is that communications are usually based on the multicast paradigm, where packets being sent by one sender are duplicated by switching elements in the network and received by multiple receivers. Using DNC on such multicast protocols requires some adaptations, since this method is restricted to the analysis of unicast communications. As detailed later, in Section 3, previous attempts for using DNC to analyze multicast communications only circumvented its current restriction. They do not provide a solution to overcome this limitation. Those approaches cannot benefit from all DNC capabilities to provide accurate end-to-end guarantees and networks designed based on them will be over-dimensioned.

We address the open issue of multicast flow analysis with DNC. We contribute two approaches that turn out to be steps generalizing existing analyses. The first one, Explicit Intermediate Bounds (EIB), is an approach where multicast flows are cut into sequences of unicast sub-flows. End-to-end performance bounds are then derived from sub-flow results. It does not require a transformation of the network, however, it amends a step to the analysis. Our second generalization finally leads to a DNC multicast feed-forward analysis. Neither transforming the network nor cutting any flows is required. Therefore, more accurate bounds are obtained since existing DNC principles can be applied in order to reduce effects such as flow multiplexing or burstiness. We numerically evaluate our proposed methods on two AFDX networks given in the literature and show that our DNC results are on par with other analytical methods or outperform them.

This paper is organized as follows: Section 2 gives a brief background on deterministic network calculus and we derive the foundation for our generalizations. In Section 3, we present related work on multicast flow analysis. Sections 4 and 5 contribute generalizations of DNC analyses for the study of multicast flow guarantees. We evaluate our approaches in Section 6 and Section 7 concludes the paper.

ACM ISBN .
DOI:

2. NETWORK CALCULUS BACKGROUND

We present in this section a brief overview of deterministic network calculus. For a more in-depth description, we refer the reader to [9] and [14].

2.1 Flow and Server Modeling

In network calculus, flows correspond to unidirectional and unicast communications between two servers. They are modeled as functions of their cumulative arrival of data. More formally, those functions belong to the following set $\mathcal{F}_0$ of non-negative, wide-sens increasing functions:

$$\mathcal{F}_0 = \{f : \mathbb{R} \to \mathbb{R}^+ \mid f(0) = 0, \forall 0 \leq s < t : f(s) < f(t)\}$$

In order to compute bounds on the flows, we are interested in the functions $A(t)$ corresponding to the data arriving in a given server s at time t, and $A'(t)$ the amount of data processed by the server at time t. Using this formalism, the following delay definition can then be derived:

Definition 1 (Flow delay). *Assume a flow with input A and crosses a server s and results in the output A'. The (virtual) delay for a data unit arriving at time t is*

$$D(t) = \inf\{\tau \geq 0 \mid A(t) \leq A'(t + \tau)\}$$

Instead of directly working with A, network calculus makes use of the concept of arrival curves, which is a function bounding the maximal arrivals of a flow:

Definition 2 (Arrival curve). *Given a flow with input A, a function $\alpha \in \mathcal{F}_0$ is an arrival curve for A iff*

$$A(t) - A(s) \leq \alpha(t - s), \forall t, s, 0 \leq s \leq t$$

Definition 3 (Service curve). *If the service provided by a server s for a given input A results in an output A', then s offers a service curve $\beta \in \mathcal{F}_0$ iff*

$$A'(t) \geq \inf_{0 \leq s \leq t}\{A(t - s) + \beta(s)\}, \forall t$$

2.2 (min, +) Algebra

Network calculus was formalized as a $(\min, +)$-algebraic framework in [9, 14], enabling an easier description of operations on flow and server descriptions.

Definition 4 ($(\min, +)$ operations). *The $(\min, +)$ convolution and deconvolution of two functions $f, g \in \mathcal{F}_0$ are defined as:*

$$\text{Convolution: } (f \otimes g)(t) = \inf_{0 \leq s \leq t}\{f(t - s) + g(s)\}$$

$$\text{Deconvolution: } (f \oslash g)(t) = \sup_{s \geq 0}\{f(t + s) - g(s)\}$$

Using those $(\min, +)$ operations, one can rewrite the previous definitions as $A' \geq A \otimes \beta$ and $A \otimes \alpha \geq A$.

Moreover, $(\min, +)$ convolution allows DNC to concatenate the service of consecutive servers $\langle 1, \ldots, n\rangle$, so-called tandems, into a single service curve:

Theorem 1 (Concatenation of servers). *Consider a single flow f crossing a tandem of servers $s_1, \ldots, s_n$ where each server s_i offers a service curve β_i. The overall service curve for f is their concatenation by convolution:*

$$\beta_i \otimes \cdots \otimes \beta_n = \bigotimes_{i=1}^{n} \beta_i$$

Given a strict service curve that guarantees a minimum output of β if data is present at a server, we lower bound the service left-over for a specific flow:

Theorem 2 (Left-over service curve). *Consider a server s that offers a strict service curve β. Let s be crossed by flows f_0 and f_1, with arrival curves α_0, respectively α_1. Then the worst-case residual resource share under arbitrary multiplexing of f_1 at s is:*

$$\beta^{\text{l.o.}f_1} = \beta \ominus \alpha_0$$

with $(\beta \ominus \alpha)(d) = \sup\{(\beta - \alpha)(u) \mid 0 \leq u \leq d\}$ denoting the non-decreasing upper closure of $(\beta - \alpha)(d)$.

Last, we use these curves to derive performance bounds.

Theorem 3 (Performance bounds [14]). *Consider a flow f with arrival curve α traversing a server s with a service curve β. The following bounds can be derived:*

$$\text{Backlog: } Q(t) \leq \sup_{s \geq 0}\{\alpha(s) - \beta(s)\} = (\alpha \oslash \beta)(0)$$

$$\text{Delay: } D(t) \leq \inf\{d \geq 0 \mid (\alpha \oslash \beta)(-d) \leq 0\}$$

$$\text{Output: } \alpha'(d) = (\alpha \oslash \beta)(d)$$

with α' being an output arrival curve for A'.

2.3 Network Analysis

Using the definitions and theorems presented above, the end-to-end performances of flows interacting on a network of servers can be computed. We call the analyzed flow *flow of interest*, abbreviated foi.

2.3.1 Tandems of Servers

The foi's path defines the sequence (tandem) of servers that defines its end-to-end delay. For bounding this delay, different methods have been proposed in the literature.

Total Flow Analysis (TFA) [14].

The TFA first computes per-server delay bounds. Each one holds for the sum of all the traffic arriving to a server, i.e., these bounds are independent of the foi. The flow's end-to-end delay bound is derived by summing up the individual server delay bounds on its path. The TFA's server-isolating approach constitutes a direct application of Theorem 3; it is known to be inferior to the following analyses [14, 18].

Separated Flow Analysis (SFA) [14].

The SFA is a direct application of other theorems: first compute the left-over service of each server on the foi's path using Theorem 2, then concatenate them using Theorem 1 and finally derive the end-to-end delay bound using Theorem 3. Deriving the end-to-end delay bound using only one service curve will consider the burst term of the foi only once, a property called Pay Burst Only Once (PBOO).

Pay Multiplexing Only Once (PMOO) [18].

The PMOO analysis first convolves the tandem of servers before subtracting the cross-traffic. Using this order, the bursts of the cross-traffic appear only a single time compared to the SFA analysis where the bursts are included at each server. Therefore, multiplexing with cross-traffic is only paid for once. However, [17] showed that the PMOO method does not necessarily outperform the SFA.

2.3.2 *Feed-forward Networks*

For more involved feed-forward networks, a procedure to combine tandem analyses to a network analysis exists. In order to integrate the analysis of multicast flows into DNC, we derive this procedure in great detail. Here, we contribute the following result: a precise structure of the steps taken by any DNC network analysis. The well-elaborated structure we establish in this section, also serves us to judge and compare different approaches to aiming for performance bounds in feed-forward networks with multicast flows.

In previous work, two basic steps of the analysis have already been identified [5]: *1)* cross-traffic arrival bounding and *2)* flow of interest performance bounding. They are tailored to a compositional unicast flow analysis. We call it the *unicastFFA* and derive its steps in unprecedented detail:

unicastFFA Step 1: Cross-traffic Arrival Bounding. The first unicastFFA step abstracts from the feed-forward network to the foi's path – a tandem of servers that can be analyzed with one of the existing procedures. In detail, this step proceeds as follows:

(i) Starting at the locations of interference with the foi, cross-flows are backtracked to their sources. This procedure derives the dependencies between the foi, its cross-flows, their cross-flows, etc., in a recursive fashion. A new instance of this sub-step is started for any cross-flow of the current cross-flow under consideration. Due to the network's feed-forward property, the recursion is guaranteed to terminate.

(ii) Next, the dependencies are converted into equations, i.e., a sequence of algebraic operations for each location of interference with the foi. They capture the worst-case transformation of flow arrivals towards foi.

(iii) Finally, the equations are solved to obtain the bounds on cross-traffic arrivals.

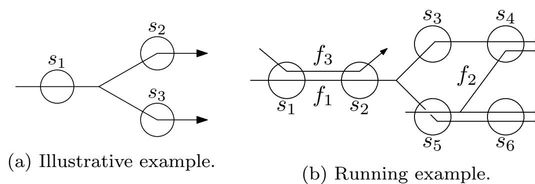
After these substeps, all cross-flows' arrivals are bounded with arrival curves (arrival bounds).

unicastFFA Step 2: foi Performance Bounding. Given the cross-traffic arrival bounds from step 1, step 2 does not need to consider the part of the network traversed by these flows nor the potentially complex interference patterns they are subject to. The foi's end-to-end delay bound in the feed-forward network is derived with a tandem analysis.

Note, that this step provides information required in the previous one. It defines the flow of interest and thus its cross-flows as well as their locations of interference used in step 1(i). This step is strongly based on the tandem analysis that, in turn, is derived from the aim to analyze a unicast flow from end to end. It is not directly applicable to the analysis of multicast flows and thus needs generalization.

2.4 Multicast Flows

As defined at the beginning of this section, flows and network analysis in network calculus have been mostly focused on the modeling of unidirectional and unicast communications. Such a model is not directly applicable to multicast network protocols, where packets are duplicated at certain points of the network in order to provide one-to-many communications as illustrated in Figure 1.



(a) Illustrative example.

(b) Running example.

Figure 1: Multicast networks.

We define the following terms for describing parts of a multicast flow:

Definition 5 (Trajectory and fork). *A trajectory of a given source-sink pair corresponds to the equivalent unicast flow going from the source to the sink. A fork corresponds to a server where packets are duplicated.*

In the following, we will provide an illustration as well as a running example. They are meant to serve two distinct purposes: On the one hand, we depict the basic idea behind the approaches to handle multicast flows with the minimal network of Figure 1a. Additionally, we will analyze the network of Figure 1b with the given approach. Analyzing flow f_2 makes this network minimal w.r.t. covering all effects relevant to DNC and multicast flows: There one multicast flow in each step of the analysis, cross-traffic arrival bounding (f_1) as well as flow of interest analysis (f_2). Moreover, a unicast flow is present and this network allows us to observe the impact of different flow analyses described earlier in Section 2.3.1 (TFA, SFA with the PBOO effect, PMOO).

3. RELATED WORK

In this related work section, we provide two DNC approaches to the analysis of multicast flows. For both, we focus on how these approaches enable the unicastFFA of the previous section to analyze networks with multicast flows.

unicastFFA Transformation: A Set of Unicast Flows

A first approach to circumvent this issue is to transform a multicast flow to a set of unicast flows. It was mentioned as a possibility to cope with multicast flows in [6]. Each trajectory will become one independent unicast flow, as illustrated in our two sample networks (Figures 2a and 2b).

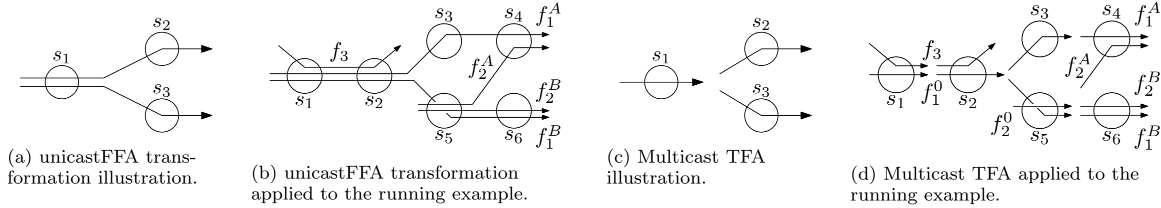
From a procedural point of view, the unicast transformation does not integrate into the unicastFFA. It only enables for using it by a preceding step that transforms the network. This step is static, i.e., it does not consider the unicastFFA's information like the flow(s) that are under analysis.

The foremost problem of this approach is its overly pessimistic assumption about resource demand of multicast flows. On common sub-paths of a multicast flows' trajectories, i.e., the servers before a fork, multiple unicast flows compete for resources. The unicastFFA thus models the worst case with mutual interference between these flows that are not present in the original network model.

On the other hand, this approach allows for the PBOO and the PMOO principle in the unicastFFA.

Multicast TFA

Grieu [11] proposes a procedure to apply the TFA presented in Section 2.3.1 in the analysis of multicast flows. It is tailored to the TFA and shares its inherent isolation of servers.



(a) unicastFFA transformation illustration.

(b) unicastFFA transformation applied to the running example.

(c) Multicast TFA illustration.

(d) Multicast TFA applied to the running example.

Figure 2: Existing DNC approaches to the multicast analysis applied to the networks presented in Figure 1.

Thus, it does not integrate into the unicastFFA for deriving delay bounds. Figures 2c and 2d depict this procedure on the illustration and the running example, respectively. Flows are cut between all servers, the arrivals are aggregated and a server-local delay bound is computed. In a second step, the server delay bounds on the trajectory of interest are summed up. As this last step is similar to the unicastFFA step 1, it inherits its decisive TFA shortcomings. I.e., neither the PMOO nor the PBOO principle are implemented and the delay bounds are known to be inaccurate.

Related Approaches

The existing DNC approaches both have significant drawbacks. Therefore, the literature created novel multicast analyses based on the DNC system description.

The *Trajectory Approach* (TA) is an adaptation to the study of network delays of the holistic approach [19]. It was originally developed to give bounds on the scheduling of tasks on a processor. The approach was initially proposed in [16] and later extended to FIFO systems in [15]. [1] applied TA to the study of avionic networks with multicast flows and showed, via numerical evaluations, that it outperforms the multicast TFA.

The *Forward End-To-End Delay Approach* (FA) has been proposed more recently in [13]. It addresses the shortcomings of the TA. Similarly to the TA, FA is also an adaptation of the holistic approach to the case of FIFO networks. [13] and [12] applied the FA to the performance evaluation of avionic networks with multicast flows and showed that this approach outperforms the multicast TFA as well.

Although FA sets its focus on the end-to-end analysis – similar to the DNC tandem analyses – neither FA nor TA have been benchmarked against a modern DNC that implements PBOO or PMOO. This can be attributed to the lack of such an analysis for multicast flows. In this paper, we will generalize multicast TFA as well as the unicastFFA in order to provide such DNC solutions and benchmarking results.

4. EXPLICIT INTERMEDIATE BOUNDS

Explicit Intermediate Bounds (EIB) analysis is the generalization of the multicast TFA analysis. We do not create an entirely new analysis (unlike TA and FA) but adapt the model such that we can analyze it with our NC tools. EIB combines the strengths of the two related approaches from above. Like the TFA, it follows the foi's trajectory of interest without additional cross-traffic assumptions. Moreover, it proceeds in tandems like the set of unicast flows. Therefore, it can benefit from both the PBOO and the PMOO principle on these tandems – unlike the server-local approach responsible for TFA's general inferiority w.r.t. delay bounds.

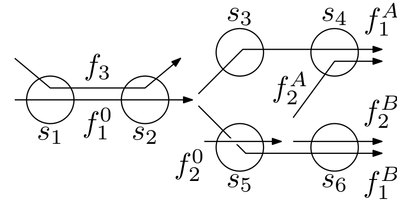


Figure 3: Application of the EIB analysis to the studied scenario: multicast flows are cut into unicast sub-flows.

We achieve this by an approach consisting of two parts:

1. A static, preceding step takes the network description and transforms it for analysis. In contrast to the multicast TFA, we cut multicast flows after forking locations only; not after every server. These are the locations of explicit intermediate bounds. In our illustrative example, the result is equal to the multicast TFA (Figure 2c). However, we transform the network into tandems that are crossed by parts of the multicast flows. This is revealed by our running example. For instance, compare the tandems $\langle 1, 2\rangle$ and $\langle 3, 4\rangle$ of multicast TFA (Figure 2d) with EIB (Figure 3).
2. The locations of cuts is static, i.e., only defined by the network, not the foi. The EIB analysis can be viewed to create a global worst case for all flows, independent of the foi. Afterwards it runs a unicastFFA in its second part.

These two steps are, however, not as separated as in the literature's multicast TFA. We integrate the EIB idea into the unicastFFA is as follows:

- Step 1: An adaptation of the cross-traffic arrival bounding is required at the locations of explicit intermediate bounds. Previously, tandems for analysis were defined by the common path of flow aggregates [6]. Using EIB, we have an additional restriction for the tandem lengths as we derive the output bound at the server a multicast flow forks. I.e., when the backtracking reaches such a server, the search for a tandem to operate on terminates. A new instance of unicastFFA step 1 is started with this server being the last one for the next tandem in the analysis.
- Step 2: The flow of interest analysis actually becomes a "trajectory of interest" analysis. Yet, for multicast flows it is not able to analyze the trajectory in an end-to-end fashion with a single left-over service curve.

A multicast flow's entire trajectory will consist of at least two tandems as it has at least one fork location to cut at. Thus, the PMOO principle cannot be implemented entirely; similar to the FIFO multiplexing tandem analysis LUDB [2].

We call this integrated procedure EIB unicastFFA; the name emphasizes the application of this specific way to obtain results with the EIB idea. Regarding the results themselves, EIB unicastFFA and EIB are synonymous as both of the above procedures yield the same performance bounds.

Analysis of the Running Example

As mentioned above, the difference between multicast TFA and EIB unicastFFA becomes apparent in more involved networks. We derive the left-over service curves for multicast flow f_2 of our running example (Figure 3) to depict the new analysis. We follow the EIB unicastFFA procedure that does not reveal a preceding EIB step.

EIB unicastFFA step 1. Bounding the delay of f_2 requires to bound the interference of multicast cross-flow f_1. The first location of interference with f_2 is at s_5 where we need to backtrack f_1^B. EIB enforces a cut after f_1's fork at server s_2 that we need to consider. We get $\alpha_5^{f_1^B} = \alpha_5^{f_1^0}$, i.e., the backtracking is stopped there. In a second instance of EIB unicastFFA step 1, f_1^0 is backtracked in order to derive its explicit intermediate bound at the output of s_2 / at the input of server s_5:

$$\alpha_5^{f_1^B} = \alpha^{f_1^0} \oslash \beta_{\langle 1,2\rangle}^{\mathrm{l.o.}f_1^0}$$

In contrast to the multicast TFA, we can benefit from either SFA/PBOO or PMOO in the derivation of $\beta_{\langle 1,2\rangle}^{\mathrm{l.o.}f_1^0}$. The according derivation that applies both alternatives is called aggregate arrival bounding, aggrAB, [6]:

$$\begin{aligned}\alpha^{f_1^0} \oslash \beta_{\langle 1,2\rangle}^{\mathrm{l.o.}f_1^0} &= \left(\alpha^{f_1^0} \oslash \left(\left(\beta_1 \ominus \alpha^{f_3}\right) \otimes \left(\beta_2 \ominus \alpha_2^{f_3}\right)\right)\right) \\ &\quad \wedge \left(\alpha^{f_1^0} \oslash \left((\beta_1 \otimes \beta_2) \ominus \alpha^{f_3}\right)\right)\end{aligned}$$

where $\alpha_2^{f_3}$ corresponds to the arrival bound of f_3 at s_2.

At the second location of interference between f_1 and f_2, server s_4, we need to backtrack trajectory f_1^A. This yields

$$\begin{aligned}\alpha_4^{f_1^A} &= \alpha_5^{f_1^B} \oslash \beta_3 \\ &= \left(\alpha^{f_1^0} \oslash \beta_{\langle 1,2\rangle}^{\mathrm{l.o.}f_1^0}\right) \oslash \beta_3 = \alpha^{f_1^0} \oslash \left(\beta_{\langle 1,2\rangle}^{\mathrm{l.o.}f_1^0} \otimes \beta_3\right)\end{aligned}$$

Note, that we cannot derive a left-over service curve $\beta_{\langle 1,2,3\rangle}^{\mathrm{l.o.}f_1^A}$ due to the cut enforced by EIB. In general, this prevents the implementing of the PMOO principle.

EIB unicastFFA step 2. Next, we bound the delay of trajectory f_2^A. Again, it cannot be done with a single, PMOO left-over service curve due to EIB's cut between s_5 and s_6. In this case, the cut means SFA is the only analysis option:

$$\begin{aligned}\beta^{\mathrm{l.o.}f_2^A} &= \beta_{\langle 5,4\rangle}^{\mathrm{l.o.}f_2^A} \\ &\text{(cut enforced by EIB, no single-tandem analysis)} \\ &= \beta_5^{\mathrm{l.o.}f_2^A} \otimes \beta_4^{\mathrm{l.o.}f_2^A} \\ &= \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_4 \ominus \alpha_4^{f_1^A}\right)\end{aligned}$$

where $\alpha_5^{f_1^B}$ and $\alpha_4^{f_1^A}$ have been derived in EIB unicastFFA step 1, the cross-traffic arrival bounding.

In the analysis of f_2, the delay bound for trajectory f_2^B remains to be bounded. Again, we need to derive the according left-over service curve that illustrates the proceedings of EIB unicastFFA:

$$\begin{aligned}\beta^{\mathrm{l.o.}f_2^B} &= \beta_{\langle 5,6\rangle}^{\mathrm{l.o.}f_2^B} \\ &\text{(cut enforced by EIB, no single-tandem analysis)} \\ &= \beta_5^{\mathrm{l.o.}f_2^B} \otimes \beta_6^{\mathrm{l.o.}f_2^B} \\ &= \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_6 \ominus \alpha_6^{f_1^B}\right)\end{aligned}$$

Most notably, f_2^B sees multi-hop interference by f_1^B but the left-over service curve derivation cannot make use of the PMOO principle. EIB inhibits an end-to-end analysis in this unicastFFA step 2, only SFA/PBOO can be applied

Theoretical Evaluation

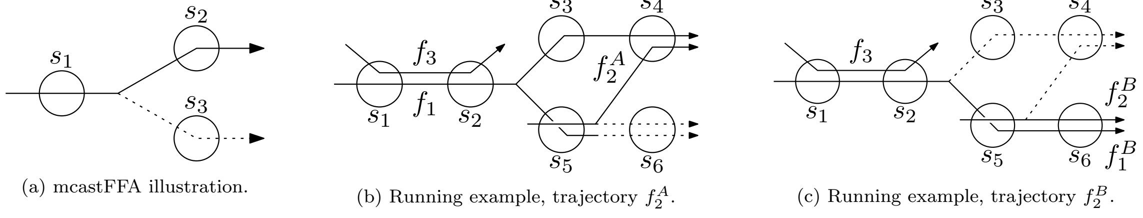
We conclude this section by a theoretical evaluation of the EIB approach against the previous DNC approaches:

- *Relation to multicast TFA:*
  The integration into the unicastFFA constitutes a generalization of the multicast TFA. On every tandem to analyze in either of the two unicastFFA steps, we can apply the TFA depicted in Section 2.3. Then, an additional intermediate step that separates the individual servers is executed and the per-server results are composed to the respective tandem result. The multicast TFA delay bounds cannot outperform those of EIB with either SFA or PMOO analysis on all tandems.
- *Relation to unicastFFA transformation:*
  In contrast to this section's EIB, the unicastFFA transformation allows for a single end-to-end left-over service curve for every trajectory of interest. I.e., it can fully benefit from the PMOO principle. However, the cuts enforced by EIB can also have a positive effect. They break the $\otimes$'s commutativity, allowing to better exploit service rates on the tandem. This is a variant of the problem shown in [17], but with a handicapped PMOO. None of the alternatives is strictly superior.

5. A MULTICAST FEED-FORWARD ANALYSIS PROCEDURE

In this section, we generalize the unicastFFA presented in Section 2.3.2 to networks with multicast flows. We call this generalized method *multicast Feed-Forward Analysis*, or mcastFFA. This allows us to make use of the knowledge only available in the unicastFFA itself. In contrast to the existing DNC approaches and the EIB analysis, no network transformation is amended to the analysis. We do not create a network-wide worst-cast setting for all flows before executing the unicastFFA. Instead, our generalization solely constructs a single flow of interest's worst case during analysis – a less pessimistic setting than the network-wide one for all flows at once. With this approach, the mcastFFA analysis obtains best results by exploiting the PMOO and PBOO principles to a larger extent than the EIB.

Figure 4a illustrates the basic idea behind our solution: If we analyze this multicast flow's trajectory crossing s_2,



(a) mcastFFA illustration. (b) Running example, trajectory f_2^A. (c) Running example, trajectory f_2^B.

Figure 4: Application of mcastFFA. The dashed lines depict parts of flows that are not considered in the current analysis.

the other trajectory crossing s_3 becomes irrelevant for the delay bound computation. We neither need to add an entire cross-flow for it nor do we require the output bound from s_1. Thus, mcastFFA can treat the multicast trajectory (or unicast flow) of interest in an end-to-end fashion and apply, for instance, the PMOO principle.

The main challenge of this approach is to reduce the network to relevant servers as well as (partial) flows and multicast flow trajectories. This may constitute considerable effort in large networks. Therefore, we present a solution that generalizes the unicastFFA analysis in order to gain from its efficiency [3]. I.e., deriving the sub-network relevant to a specific foi is part of the analysis proceedings, not covered by a separate step. Our mcastFFA solution is mainly based on the newly derived sub-step 1(i) of the unicastFFA: backtracking of dependencies.

In this step, dependencies of a flow on others are derived by traversing the network in the opposite direction of links [5]. The entire unicastFFA starts this procedure with the flow of interest. Our mcastFFA will iterate over all n trajectory of interest and execute separate analyses. In case of a unicast flow, we get $n = 1$; for multicast flows n equals the amount of trajectories (source/sink-pairs). Multicast cross-flows are traversed backwards, too, such that their fork locations do not enforce to cut the tandem to analyze; the relevant trajectory of the cross-flow is known and can be treated similar to a unicast cross-flow. Namely, the mcastFFA is a generalization of the known unicastFFA and thus it operates on longer tandems than EIB.

Whereas the EIB required to explicitly consider each location a multicast flow forks, the mcastFFA implicitly restricts the analysis to the trajectory relevant for the analysis. After the backtracking, we know the entire sub-network whose servers and (partial) flows appear in the analysis equation of unicastFFA step 1(ii). I.e., we rely on the detailed understanding of unicastFFA that we derived in Section 2.3.2.

Analysis of the Running Example

We will derive the left-over service curves for f_2's trajectories in order to compare them against the EIB unicastFFA. For brevity, we restrict the depiction to f_2^A's cross-traffic arrival bounding (mcastFFA step 1, Figure 4b) and f_2^B's delay bounding (mcastFFA step 2, Figure 4c). These derivations depict the crucial improvement of mcastFFA's proceedings in both of the analysis steps. They point out the reduction of the network and the increased tandem lengths.

mcastFFA step 1. We consider f_2^A's cross-traffic arrival bounding. Backtracking will be "local" to a single trajectory of a multicast cross-flow. In our example, we finally have established the possibility to apply the PMOO-principle when computing f_1^A's aggregate arrival bound aggrAB at server s_4 [6]. See $\alpha_4^{f_1^A}$ in the follwoing left-over service curve derivation we require to bound cross-traffic arrivals:

$$\beta^{\text{l.o.}f_2^A} = \beta_{\langle 5,4\rangle}^{\text{l.o.}f_2^A}$$

(only single-hop interference so cutting is fine)

$$\begin{aligned}
&= \beta_5^{\text{l.o.}f_2^A} \otimes \beta_4^{\text{l.o.}f_2^A} \\
&= \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_4 \ominus \alpha_4^{f_1^A}\right) \\
&= \left(\beta_5 \ominus \left(\alpha^{f_1} \oslash \beta_{\langle 1,2\rangle}^{\text{l.o.}f_1}\right)\right) \otimes \left(\beta_4 \ominus \left(\alpha^{f_1} \oslash \beta_{\langle 1,2,3\rangle}^{\text{l.o.}f_1^A}\right)\right)
\end{aligned}$$

A cut of $\beta_{\langle 1,2,3\rangle}^{\text{l.o.}f_1^A}$ into $\beta_{\langle 1,2\rangle}^{\text{l.o.}f_1} \otimes \beta_3^{\text{l.o.}f_1^A}$ was needed in the EIB analysis, meaning that PMOO could not be implemented.

This advantage is also depicted in Figure 4b where f_1 retains its multicast shape in the mcastFFA's point of view.

mcastFFA step 2. For the second trajectory of f_2, f_2^B, our mcastFFA derives $\beta^{\text{l.o.}f_2^B} = \beta_{\langle 5,6\rangle}^{\text{l.o.}f_2^B}$. Again, we are not enforced to cut this trajectory's path (see Figure 4c) and in contrast to EIB we can apply alternative tandem analyses:

- SFA/PBOO:

$$\beta^{\text{l.o.}f_2^B} = \beta_{\langle 5,6\rangle}^{\text{l.o.}f_2^B}$$

(cut enforced by SFA, no single-tandem analysis)

$$\begin{aligned}
&= \beta_5^{\text{l.o.}f_2^B} \otimes \beta_6^{\text{l.o.}f_2^B} \\
&= \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_6 \ominus \alpha_6^{f_1^B}\right) \\
&= \left(\beta_5 \ominus \left(\alpha^{f_1} \oslash \beta_{\langle 1,2\rangle}^{\text{l.o.}f_1}\right)\right) \otimes \left(\beta_6 \ominus \left(\alpha^{f_1} \oslash \beta_{\langle 1,2,5\rangle}^{\text{l.o.}f_1^B}\right)\right)
\end{aligned}$$

Note, that the actual trajectory of the cross-flow, f_1 or f_1^B, was automatically chosen correctly by the backtracking. Moreover, note the contrast to EIB: We can derive f_1^B's arrivals at s_6 with an end-to-end left-over service curve that, in turn, can make use of aggrAB.

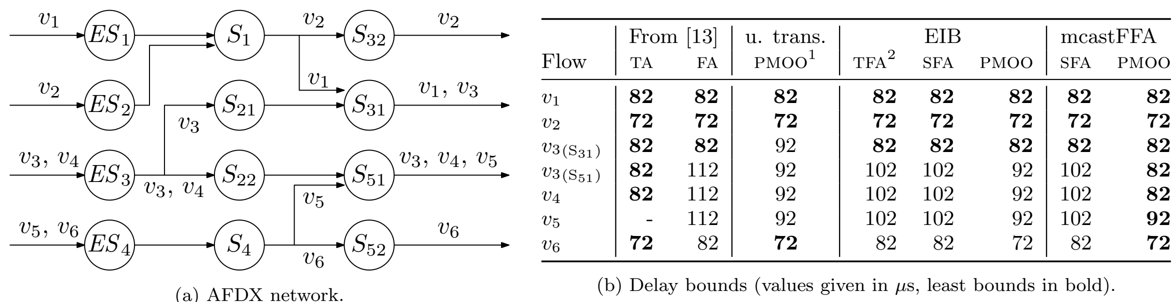
- PMOO:

$$\beta^{\text{l.o.}f_2^B} = \beta_{\langle 5,6\rangle}^{\text{l.o.}f_2^B}$$

(there is no enforced cut)

$$\begin{aligned}
&= (\beta_5 \otimes \beta_6) \ominus \alpha_5^{f_1} \\
&= (\beta_5 \otimes \beta_6) \ominus \left(\alpha^{f_1} \oslash \beta_{\langle 1,2\rangle}^{\text{l.o.}f_1}\right)
\end{aligned}$$

where $\beta_{\langle 1,2\rangle}^{\text{l.o.}f_1}$ can be computed either applying the left-over service curve derivation of SFA/PBOO or PMOO.



(a) AFDX network.

Flow	From [13] TA	From [13] FA	u. trans. PMOO[1]	EIB TFA[2]	EIB SFA	EIB PMOO	mcastFFA SFA	mcastFFA PMOO
v_1	**82**	**82**	**82**	**82**	**82**	**82**	**82**	**82**
v_2	**72**	**72**	**72**	**72**	**72**	**72**	**72**	**72**
$v_{3(S_{31})}$	**82**	**82**	92	**82**	**82**	**82**	**82**	**82**
$v_{3(S_{51})}$	**82**	112	92	102	102	92	102	**82**
v_4	**82**	112	92	102	102	92	102	**82**
v_5	-	112	92	102	102	92	102	**92**
v_6	**72**	82	**72**	82	82	72	82	**72**

(b) Delay bounds (values given in μs, least bounds in bold).

Figure 5: Simple AFDX network evaluation of [13], extended with DNC's EIB and mcastFFA delay bounds.

This derivation is illustrated in Figure 4c. Compared to Figure 3, we indeed notice the longer tandem for the second trajectory of f_2.

Theoretical Evaluation

We conclude this section by a theoretical evaluation of mcastFFA against the related DNC approaches:

- *Relation to unicastFFA (Section 2.3.2):* The mcastFFA is a generalization of the unicastFFA. Analysis of unicast flows in either of the two steps remains unaffected (see f_3 in the running example).
- *Relation to unicastFFA transformation (Section 3):* Like the unicastFFA transformation, the mcastFFA is able to derive a PMOO end-to-end left-over service curve. However, it does so without the additional cross-traffic assumptions introduced by the unicastFFA transformation. I.e., there are less cross-flows to consider in the analysis, left-over service curves will be larger and delay bounds will be smaller. Thus, mcastFFA outperforms unicastFFA transformation.
- *Relation to EIB unicastFFA:* In comparison to EIB, we gained the ability to operate on end-to-end tandems. This constitutes increased flexibility to cut this tandem during the analysis: Our mcastFFA is compatible with SFA/PBOO, PMOO, aggrAB, or [3] for best attainable left-over service curves. This best solution to cut a tandem and combine subtandem results might coincide with EIB's enforced alternative, i.e., mcastFFA is indeed a generalization of EIB unicastFFA.

Before evaluating our contributions, let us briefly clarify their impact on the server backlog bound Q presented in Theorem 3. Deriving these bounds requires the arrival bounds of all flows at a server. I.e., in the DNC analysis procedures, (EIB) unicastFFA and mcastFFA, step 1 is crucial for the result accuracy; step 2 is not required. As shown with the running example, we improved of cross-traffic arrival bounding in case there are multicast flows present. Thus, backlog bounds are also improved by our contribution.

6. NUMERICAL EVALUATION

We have shown in Sections 4 and 5 that our proposed approaches are superior to the previous network calculus ones, presented as related work in Section 3. We investigate now in this section the gains in terms of accuracy of end-to-end delay bounds via a numerical evaluation. We study the two AFDX networks presented in [13] and [12]. This allows us to benchmark our proposed approaches with the TA and FA since numerical results are given in the literature. Note, that [12] extends the TA and FA by a grouping property that accounts for serialization of packetized flows when crossing links. We leave its implementation in the generalized DNC solutions, based on [8], to future work and restrict our comparison to the non-serialized results. This also allows us to use the range of established SFA/PBOO and PMOO $\beta^{\text{l.o.}}$ derivations implemented in the DiscoDNC tool [4] as well as the aggrAB arrival bounding. I.e., we inherit the DiscoDNC assumptions of a fluid model and curves that can be decomposed into a set of either token buckets or rate latencies.

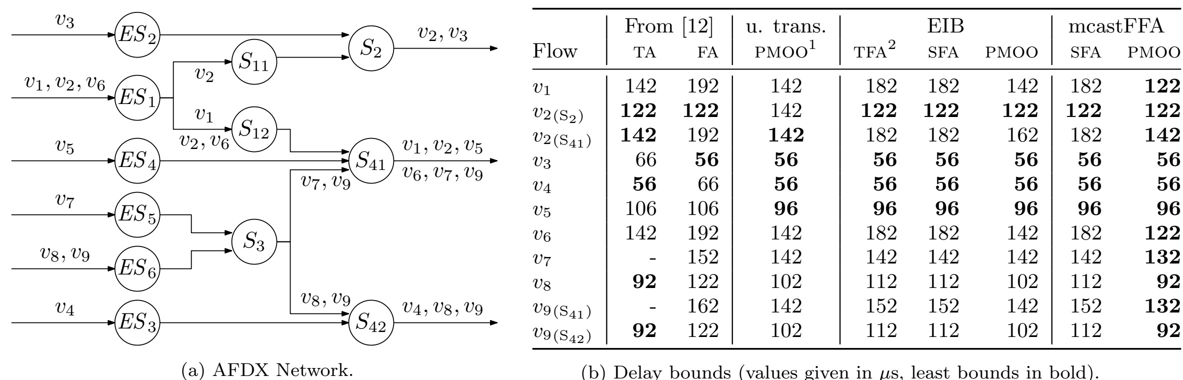
The first network, illustrated in Figure 5a, is a simple AFDX scenario with only one multicast flow (v_3) and a simple flow interference pattern. The second network, illustrated in Figure 6a, is a more complex AFDX scenario with two multicast flows (v_2 and v_9). Numerical results on the end-to-end delay bounds of the different flows are shown in Figures 5b and 6b, respectively. Key observations w.r.t. the performance of DNC analyses confirm our conjectures:

- mcastFFA with PMOO produces gains of 8.74% and 13.08% respectively compared to the multicast TFA.
- mcastFFA produces more accurate bounds than the EIB analysis, since it can operate on longer tandems.
- For some flows, all results are equal. These are simplistic cases that become less in the larger network.

Next, we compare mcastFFA to TA and FA. We observe that mcastFFA results are never inferior to these contenders. Moreover, cases of equal results often coincide with the simplistic ones where even mutlicast TFA is competitive. This is especially visible in the second scenario with less competitive TA and FA bounds. A maximum gain of 5.86% compared to TA and 18.58% compared to FA is achieved in this small AFDX scenario. AFDX networks as deployed in existing Airbus aircraft are already far bigger and more involved than the ones of Figures 5a and 6a. They consist of ∼1000 multicast flows (virtual links, VLs) that have an average of ∼6.5 trajectories per VL. Therefore, the improvements we achieve with DNC's PMOO in conjunction with mcastFFA is expected to be considerably larger in practice.

[1]unicastFFA transformation approach with the stated PMOO end-to-end left-over service curve derivation.

[2]Remember, that EIB with TFA corresponds to the multicast TFA analysis presented as related work in Section 3.



(a) AFDX Network.

Flow	From [12] TA	From [12] FA	u. trans. PMOO[1]	EIB TFA[2]	EIB SFA	EIB PMOO	mcastFFA SFA	mcastFFA PMOO
v_1	142	192	142	182	182	142	182	**122**
$v_{2(S_2)}$	**122**	**122**	142	**122**	**122**	**122**	**122**	**122**
$v_{2(S_{41})}$	**142**	192	**142**	182	182	162	182	**142**
v_3	66	**56**	**56**	**56**	**56**	**56**	**56**	**56**
v_4	**56**	66	**56**	**56**	**56**	**56**	**56**	**56**
v_5	106	106	**96**	**96**	**96**	**96**	**96**	**96**
v_6	142	192	142	182	182	142	182	**122**
v_7	-	152	142	142	142	142	142	**132**
v_8	**92**	122	102	112	112	102	112	**92**
$v_{9(S_{41})}$	-	162	142	152	152	142	152	**132**
$v_{9(S_{42})}$	**92**	122	102	112	112	102	112	**92**

(b) Delay bounds (values given in μs, least bounds in bold).

Figure 6: More complex AFDX network evaluation of [12], extended with DNC's EIB and mcastFFA delay bounds.

7. CONCLUSION

In this paper, we tackled the problem of analyzing multicast flows with deterministic network calculus. DNC was tailored to the analysis of unicast flows – a property that was assumed to invariantly hold. Therefore, previous approaches for the DNC analysis of multicast flows tried to adjust to this restriction by, e.g., pessimistic re-modeling of the network. This lead to inaccurate performance bounds and the development of alternative, non-DNC analyses to derive multicast flow guarantees. In contrast, we generalized DNC unicast feed-forward analysis to a multicast one.

We took two crucial steps to achieve this, both constituting an analysis approach of their own: the EIB analysis and the mcastFFA. In theoretical and numerical evaluations we showed that this paper contributes a single best DNC analysis for multicast flows, the mcastFFA. Not only does it outperform any other DNC approach, the evaluation of AFDX scenarios from the literature also shows that DNC achieves at least the results of competing analyses (Trajectory Approach and Forward Analysis). Moreover, the presented mcastFFA has the flexibility to be combined with any DNC tandem analysis and improvement thereof. For instance, [7], [3], FIFO multiplexing service analysis [2] or packetization [8] can tighten guarantees in AFDX networks.

8. REFERENCES

[1] H. Bauer, J. Scharbarg, and C. Fraboul. Applying and Optimizing Trajectory approach for performance evaluation of AFDX avionics network. In *Proc. IEEE ETFA*, 2009.

[2] L. Bisti, L. Lenzini, E. Mingozzi, and G. Stea. Numerical Analysis of Worst-Case End-to-End Delay Bounds in FIFO Tandem Networks. *Springer Real-Time Systems Journal*, 2012.

[3] S. Bondorf, P. Nikolaus, and J. B. Schmitt. Delay bounds in feed-forward networks – a fast and accurate network calculus solution. *arXiv:1603.02094*, 2016.

[4] S. Bondorf and J. B. Schmitt. The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus. In *Proc. EAI ValueTools*, 2014.

[5] S. Bondorf and J. B. Schmitt. Boosting Sensor Network Calculus by Thoroughly Bounding Cross-Traffic. In *Proc. IEEE INFOCOM*, 2015.

[6] S. Bondorf and J. B. Schmitt. Calculating Accurate End-to-End Delay Bounds – You Better Know Your Cross-Traffic. In *Proc. EAI ValueTools*, 2015.

[7] S. Bondorf and J. B. Schmitt. Improving Cross-Traffic Bounds in Feed-Forward Networks – There is a Job for Everyone. In *Proc. GI/ITG MMB & DFT*, 2016.

[8] M. Boyer and P. Roux. A common framework embedding network calculus and event stream theory. *hal-01311502*, 2016. Working paper or preprint.

[9] C.-S. Chang. *Performance Guarantees in Communication Networks*. Springer, 2000.

[10] F. Geyer and G. Carle. Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches. *IEEE Communications Magazine*, 2016.

[11] J. Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, INPT, 2004.

[12] G. Kemayo, N. Benammar, F. Ridouard, H. Bauer, and P. Richard. Improving AFDX End-to-End delays analysis. In *Proc. of IEEE ETFA*, 2015.

[13] G. Kemayo, F. Ridouard, H. Bauer, and P. Richard. A Forward end-to-end delays Analysis for packet switched networks. In *Proc. of RTNS*, 2014.

[14] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[15] S. Martin and P. Minet. Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class. In *Proc. of IPDPS*, 2006.

[16] J. Migge. *L'ordonnancement sous contraintes temps-réel un modèle à base de trajectoires*. PhD thesis, INRIA Sophia Antipolis, 1999.

[17] J. B. Schmitt, F. A. Zdarsky, and M. Fidler. Delay Bounds under Arbitrary Multiplexing: When Network Calculus Leaves You in the Lurch ... In *Proc. IEEE INFOCOM*, 2008.

[18] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic. Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once. In *Proc. of GI/ITG MMB*, 2008.

[19] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 1994.

A.1.2 Deterministic Network Calculus Analysis of Multicast Flows

This work was published in *Systems Modeling: Methodologies and Tools*, 2019 [20].

Chapter 10
Deterministic Network Calculus Analysis of Multicast Flows*

Steffen Bondorf† and Fabien Geyer‡

Abstract Guaranteeing performance bounds of data flows is an essential part of network engineering and certification of networks with real-time constraints. A prevalent analytical method to derive guarantees for end-to-end delay and buffer size is Deterministic Network Calculus (DNC). Due to the DNC system model, one decisive restriction is that only unicast flows can be analyzed. Previous attempts to analyze networks with multicast flows circumvented this restriction instead of overcoming it. E.g., they replaced the system model with an overly-pessimistic one that consists of unicast flows only. Such approaches impair modeling accuracy and thus inevitably result in inaccurate performance bounds.

In this chapter, we approach the problem of multicast flows differently. We start from the existing DNC analysis procedure, the unicast feed-forward analysis, and generalize it to a multicast feed-forward analysis. To that end, we contribute a novel analysis procedure that leaves the network model containing multicast flows unchanged, preserves its accuracy, allows for DNC principles such as pay multiplexing only once, and therefore derives more accurate performance bounds than existing approaches.

Key words: Delay bounds, deterministic network calculus, feed-forward networks, multicast flows

---

Steffen Bondorf
National University of Singapore (NUS), Singapore, e-mail: bondorf@comp.nus.edu.sg

Fabien Geyer
Chair of Network Architectures and Services, Technical University of Munich (TUM), D-85748 Garching b. München, Germany, e-mail: fgeyer@net.in.tum.de

* This chapter is an extended version of [4]: Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows, In: Proc. of EAI ValueTools (2016).

† Part of this work has been conducted at the Distributed Computer Systems Lab, TU Kaiserslautern (TUK), D-67663 Kaiserslautern, Germany, with support of the Carl Zeiss Foundation.

‡ Part of this work has been conducted at Airbus Group Innovations.

Steffen Bondorf and Fabien Geyer

10.1 Introduction

Distributed embedded electronic applications communicating via packet networks have become the norm in various industries such as automotive, avionic or automation. In such industrial applications, real-time constraints on packet delay and jitter are usually required in order to ensure the specified processes behavior. Due to certification of systems as well as reliability demands, formal methods are applied to validate these timing constraints. They allow for hard guarantees via upper bounds. While different analytical methods have been proposed in the literature, Deterministic Network Calculus (DNC) established itself as common method to analyze asynchronous communications in packet networks. A concrete example of this is Avionic Full-Duplex Ethernet (AFDX), a communication technology based on Ethernet and already deployed in avionic systems. Network calculus has proven to be a key method for the certification of deterministic properties of networks used for fly-by-wire [15].

An important property of those industrial networks is that communications are usually based on the multicast paradigm, where packets being sent by one sender are duplicated by switching elements in the network and received by multiple receivers. Using DNC on such multicast protocols requires some adaptations, since this method is restricted to the analysis of unicast communications. As detailed later, in Section 10.3, previous attempts for using DNC to analyze multicast communications only circumvented its current restriction. They do not provide a solution to overcome this limitation and cannot benefit from all DNC capabilities to provide accurate end-to-end guarantees.

We address the open issue of multicast flow analysis with DNC. We contribute an approach generalizing the known unicast feed-forward analysis (unicastFFA) – the DNC multicast feed-forward analysis (mcastFFA). Compared to existing approaches, more accurate bounds are obtained since advanced DNC principles can be applied in order to reduce, for instance, overly pessimistic assumptions on flow multiplexing. We numerically evaluate our proposed methods on two AFDX networks and show that our DNC results are on par with other analytical methods or outperform them.

This chapter is organized as follows: Section 10.2 presents background on DNC modeling and unicast analysis. In Section 10.3, we present related work on multicast flow performance analysis. Section 10.4 contributes a generalization of DNC unicastFFA for the study of multicast flow guarantees. We evaluate our approach in Section 10.5. Section 10.6 concludes the chapter and provides an outlook.

10.2 Deterministic Network Calculus Background

Deterministic Network Calculus models resources as bounding functions and provides (min,+)-algebraic operations to derive performance bounds from these. We

provide the basic theory applied in this chapter. For a comprehensive description, we refer the reader to [14] and [10]. Bounding functions cumulatively model arrivals or service in interval time. These belong to the set $\mathscr{F}_0$ of non-negative, wide-sens increasing functions:

$$\mathscr{F}_0 = \{f : \mathbb{R} \to \mathbb{R}^+ \mid f(0) = 0, \forall 0 \leq s < t : f(s) < f(t)\}$$

DNC makes use of the concept of arrival curves, which is a function bounding the maximal arrivals of a flow:

Definition 1 (Arrival curve). Given a flow with input A, a function $\alpha \in \mathscr{F}_0$ is an arrival curve for A iff

$$A(t) - A(s) \leq \alpha(t-s), \forall t, s, 0 \leq s \leq t$$

Minimum service is bounded in a similar way. It is based on the relation between data input and output.

Definition 2 (Service curve). If the service provided by a server s for a given input A results in an output A', then s offers a service curve $\beta \in \mathscr{F}_0$ iff

$$A'(t) \geq \inf_{0 \leq s \leq t} \{A(t-s) + \beta(s)\}, \forall t$$

The DNC analysis relies on two basic (min,+)-algebraic operations:

Definition 3 ($(\min,+)$ operations). The $(\min,+)$ convolution and deconvolution of two functions $f, g \in \mathscr{F}_0$ are defined as:

$$\text{Convolution: } (f \otimes g)(t) = \inf_{0 \leq s \leq t} \{f(t-s) + g(s)\}$$
$$\text{Deconvolution: } (f \oslash g)(t) = \sup_{s \geq 0} \{f(t+s) - g(s)\}$$

Using these operations, the above definitions translate to $A \otimes \alpha \geq A$ and $A' \geq A \otimes \beta$. Moreover, these operations are used to derive performance bounds.

Theorem 1 (Performance bounds [10]). *Consider a flow f with arrival curve α traversing a server s with a service curve β. The following bounds can be derived:*

$$\textit{Backlog: } Q(t) \leq \sup_{u \geq 0} \{\alpha(u) - \beta(u)\} = (\alpha \oslash \beta)(0)$$
$$\textit{Delay: } D(t) \leq \inf\{d \geq 0 \mid (\alpha \oslash \beta)(-d) \leq 0\}$$
$$\textit{Output: } \alpha'(d) = (\alpha \oslash \beta)(d)$$

with α' being an output arrival curve for A'.

In advanced network analysis, two further operations are relevant:

Theorem 2 (Concatenation of servers). *Consider a single flow f crossing a tandem of servers $s_1, \ldots, s_n$ where each server s_i offers a service curve β_i. The overall service curve for f is their concatenation by convolution:*

$$\beta_i \otimes \cdots \otimes \beta_n = \bigotimes_{i=1}^{n} \beta_i$$

Given a strict service curve that guarantees a minimum output of β if data is present at a server, we lower bound the service left-over for a specific flow:

Theorem 3 (Left-over service curve). *Consider a server s that offers a strict service curve β. Let s be crossed by flows f_0 and f_1, with arrival curves α_0, respectively α_1. Then the worst-case residual resource share under arbitrary multiplexing of f_1 at s is:*

$$\beta^{\mathrm{l.o.}f_1} = \beta \ominus \alpha_0$$

with $(\beta \ominus \alpha)(d) = \sup\{(\beta - \alpha)(u) \mid 0 \leq u \leq d\}$ denoting the non-decreasing upper closure of $(\beta - \alpha)(d)$.

10.2.1 Network Analysis

Using the definitions and theorems presented above, the end-to-end performances of flows interacting on a network of servers can be computed. We call the analyzed flow *flow of interest*, abbreviated foi.

10.2.1.1 Tandems of Servers

The foi's path defines the sequence (tandem) of servers that defines its end-to-end delay. The literature proposes different methods to bound this delay.

Total Flow Analysis (TFA) [10]
The TFA first computes per-server delay bounds. Each one holds for the sum of all the traffic arriving to a server, i.e., these bounds are independent of the foi. The flow's end-to-end delay bound is derived by summing up the individual server delay bounds on its path. The TFA's server-isolating approach constitutes a direct application of Theorem 1; it is known to be inferior to the following analyses [10, 23].

Separated Flow Analysis (SFA) [10]
The SFA is a direct application of other theorems: first compute the left-over service of each server on the foi's path using Theorem 3, then concatenate them using Theorem 2 and finally derive the end-to-end delay bound using Theorem 1. Deriving the end-to-end delay bound using only one service curve will consider the burst term of the foi only once, a property called Pay Burst Only Once (PBOO).

Pay Multiplexing Only Once (PMOO) [23]
The PMOO analysis first convolves the tandem of servers before subtracting the cross-traffic. Using this order, the bursts of the cross-traffic appear only a single time

compared to the SFA analysis where the bursts are included at each server. Therefore, multiplexing with cross-traffic is only paid for once. However, [22] showed that the PMOO method does not necessarily outperform the SFA.

10.2.1.2 Feed-forward Networks

For more complex feed-forward networks, a procedure to combine tandem analyses to a network analysis exists, the unicastFFA. In order to integrate the analysis of multicast flows into DNC, we outline here the structured steps taken by any DNC feed-forward analysis. This structure also serves us to judge and compare different approaches that aim for accurate performance bounds on multicast flows. In previous work, two basic steps of the analysis have already been identified [7]:

unicastFFA Step 1: Cross-traffic Arrival Bounding
The first unicastFFA step abstracts from the feed-forward network to the foi's path – a tandem of servers that can be analyzed with one of the existing procedures. In detail, this step proceeds as follows:

(i) Starting at the locations of interference with the foi, cross-flows are backtracked to their sources. This procedure derives the dependencies between the foi, its cross-flows, their cross-flows, etc., in a recursive fashion. A new instance of this sub-step is started for any cross-flow of the current cross-flow under consideration. Due to the network's feed-forward property, the recursion is guaranteed to terminate.
(ii) Next, the dependencies are converted into equations, i.e., a sequence of algebraic operations for each location of interference with the foi. They capture the worst-case transformation of flow arrivals towards foi.
(iii) Finally, the equations are solved to obtain the bounds on cross-traffic arrivals.

After these substeps, all cross-flows' arrivals are bounded with arrival curves (arrival bounds).

unicastFFA Step 2: foi Performance Bounding
Given the cross-traffic arrival bounds from step 1, step 2 does not need to consider the part of the network traversed by these flows nor the potentially complex interference patterns they are subject to. The foi's end-to-end delay bound in the feed-forward network is derived with a tandem analysis.

Note, that this step provides information required in the previous one. It defines the flow of interest and thus its cross-flows as well as their locations of interference used in step 1(i). This step is strongly based on the tandem analysis that, in turn, is derived with the goal to analyze a unicast flow from end to end. It is not directly applicable to the analysis of multicast flows and thus needs generalization.

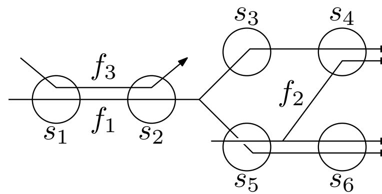


Fig. 10.1: Running example.

10.2.2 Multicast Flows

As mentioned above, flow and network analysis in network calculus have been mostly focused on the modeling of unidirectional and unicast communications. Such a model is not directly applicable to multicast network protocols, where packets are duplicated at certain points of the network in order to provide one-to-many communications as illustrated in Figure 10.1. We define the following terms for describing parts of a multicast flow:

Definition 4 (Trajectory and Fork). A trajectory of a given source-sink pair corresponds to the equivalent unicast flow going from the source to the sink. A fork corresponds to a server where packets are duplicated.

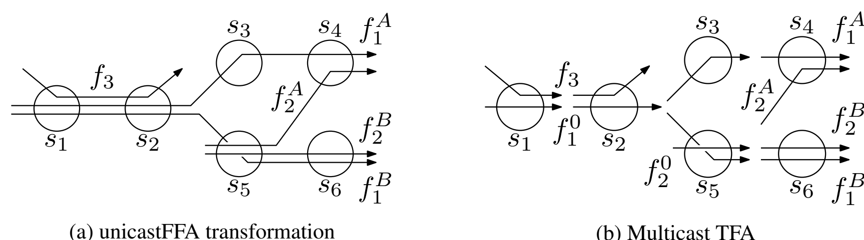
In the following, we will analyze the network of Figure 10.1 with the given approach. We focus here on the analysis of f_2, which covers all effects relevant to DNC and multicast flows: There is one multicast flow in each step of the unicast-FFA, cross-traffic arrival bounding (f_1) as well as flow of interest analysis (f_2). Moreover, a unicast flow is present and this network allows us to observe direct application of the different DNC methods described in Section 10.2.1.1, namely TFA, SFA (PBOO effect), and PMOO.

10.3 Related Work

We present three DNC approaches to analyze multicast flows. We focus on how these approaches enable the unicastFFA of the previous section to analyze networks with multicast flows. This work reveals that neither of these approaches constitutes a multicast feed-forward analysis.

unicastFFA Transformation: A Set of Unicast Flows
A first approach to circumvent the issues arising from multicast flows. Each trajectory will become one independent unicast flow, as illustrated in our sample network (Figure 10.2a) and mentioned in [8].

From a procedural point of view, the unicast transformation does not integrate into the unicastFFA. It only enables for using it by a preceding step that transforms



(a) unicastFFA transformation

(b) Multicast TFA

Fig. 10.2: Existing DNC approaches to the multicast analysis applied to the network presented in Figure 10.1.

the network. This step is static, i.e., it does not consider the unicastFFA's information like the flow(s) that are under analysis.

The foremost problem of this approach is its overly pessimistic assumption about resource demand of multicast flows. On common sub-paths of a multicast flows' trajectories, i.e., the servers before a fork, multiple unicast flows compete for resources. The unicastFFA thus models the worst case with mutual interference between these flows that are not present in the original network model. On the other hand, this approach allows for the PBOO and the PMOO principle in the unicastFFA.

Multicast TFA

Grieu [16] proposes a procedure to apply the TFA presented in Section 10.2.1.1 in the analysis of multicast flows. It is tailored to the TFA and shares its inherent isolation of servers. Thus, it does not integrate into the unicastFFA for deriving delay bounds. Figure 10.2b depicts this procedure on the running example network. Flows are cut between all servers, the arrivals are aggregated and a server-local delay bound is computed. In a second step, the server delay bounds on the trajectory of interest are summed up. As this last step is similar to the unicastFFA step 1, it inherits its decisive TFA shortcomings. I.e., neither the PMOO nor the PBOO principle are implemented and the delay bounds are known to be inaccurate.

Explicit Intermediate Bounds (EIB)

An extension of multicast TFA is presented in [4]. The authors propose a different step preceding the unicastFFA analysis. Instead of a per-server delay analysis, it analyzes the tandems of servers between a multicast flow's forks. I.e., a multicast flow is transformed into a set of sub-trajectories. These can then be analyzed individually by computing the left-over service curve on this tandem of servers. Thus, the PBOO as well as the PMOO principle can be applied. In a second step, the analyzed flow's output bounds from all sub-trajectories are derived using their left-over service curves. They are explicitly used as arrival curves after the fork locations at the end of sub-trajectories. Therefore, the approach called Explicit Intermediate Bounds (EIB). Figure 10.3 illustrates the EIB's sub-trajectory approach. Note, that the approach cannot implement the PBOO or the PMOO principle on an entire trajectory,

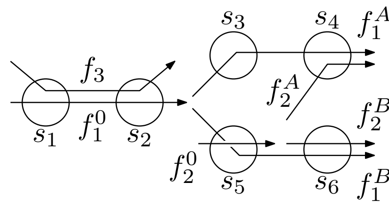


Fig. 10.3: Application of EIB: multicast flows are cut into unicast sub-trajectories.

even though the foi's left-over service curves will be convolved to attain a valid end-to-end left-over service curve for a trajectory. Moreover note, that deriving the left-over service curves required for EIB will itself result in an EIB analysis.

Non-Network Calculus Approaches

Current DNC approaches have significant drawbacks such that competing multicast analyses that build on the same modeling as DNC have been proposed.

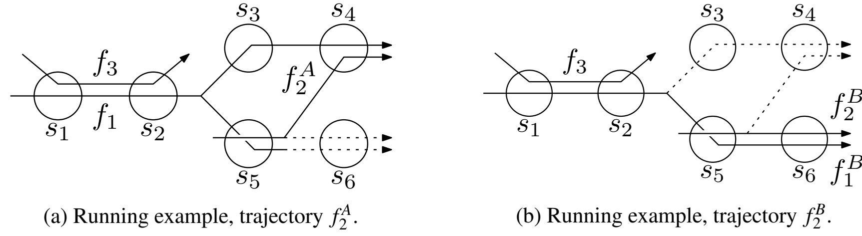
The *Trajectory Approach* (TA) is an adaptation to the study of network delays of the holistic approach [24]. It was originally developed to give bounds on the scheduling of tasks on a processor. The approach was initially proposed in [21] and later extended to FIFO systems in [20]. [2] applied TA to the study of avionic networks with multicast flows and showed, via numerical evaluations, that it outperforms the multicast TFA.

The *Forward End-To-End Delay Approach* (FA) has been proposed more recently in [18]. It addresses the shortcomings of the TA. Similarly to the TA, FA is also an adaptation of the holistic approach to the case of FIFO networks. [18] applied the FA to the performance evaluation of avionic networks with multicast flows and showed that this approach outperforms the multicast TFA as well.

Although FA sets its focus on the end-to-end analysis – similar to the DNC tandem analyses – neither FA nor TA have been benchmarked against a modern DNC that implements PBOO or PMOO. This can be attributed to the lack of such an analysis for multicast flows. We will provide such benchmarking results in Section 10.5.

10.4 A Multicast Feed-forward Analysis Procedure

In this section, we generalize the unicastFFA presented in Section 10.2.1.2 to networks with multicast flows. We call this generalized method *multicast Feed-Forward Analysis*, or mcastFFA. This allows us to make use of the knowledge only available in the unicastFFA itself. In contrast to the existing DNC approaches and the EIB analysis, no network transformation is amended to the analysis. We do not create a network-wide worst-cast setting for all flows before executing the feed-



(a) Running example, trajectory f_2^A. (b) Running example, trajectory f_2^B.

Fig. 10.4: Application of mcastFFA. The dashed lines depict parts of flows that are not considered in the current analysis.

forward analysis. Instead, our generalization solely constructs a single flow of interest's worst case during analysis – a less pessimistic setting than the static approaches constructing network-wide one for all flows simultaneously. With our approach, the mcastFFA analysis obtains best results by exploiting the PMOO principle end-to-end.

Figure 10.4b illustrates the basic idea behind our solution: If we analyze this multicast flow's trajectory crossing s_5, the other trajectory crossing s_3 becomes irrelevant for the delay bound computation. We neither need to add an entire cross-flow for it nor do we require the output bound from s_1 and s_2. Thus, mcastFFA can treat the multicast trajectories (or unicast flows) of interest in an end-to-end fashion.

The main challenge of this approach is to reduce the network to relevant servers as well as (partial) flows and multicast flow trajectories. This may constitute considerable effort in large networks. Therefore, we present a solution that generalizes the unicastFFA analysis in order to gain from its efficiency [5]. I.e., deriving the sub-network relevant to a specific foi is integral part of the analysis proceedings.

Our mcastFFA solution is mainly based on unicastFFA sub-step 1(i): backtracking of dependencies. Dependencies of flows on others are identified by traversing the network in the opposite direction of links [7]. The entire unicastFFA starts this procedure with the flow of interest. Our mcastFFA will iterate over all n trajectory of interest and execute separate analyses. In case of a unicast flow, we get $n = 1$; for multicast flows n equals the amount of trajectories. Multicast cross-flows are traversed backwards, too, such that their fork locations do not enforce to cut the tandem to analyze; the relevant trajectory of the cross-flow is known and can be treated similar to a unicast cross-flow. The mcastFFA is a generalization of the known unicastFFA. It implicitly restricts the analysis to the trajectory relevant for the analysis. After the backtracking, we know the entire sub-network whose servers and (partial) flows appear in the analysis equation of unicastFFA step 1(ii).

Analysis of the Running Example

We will derive the left-over service curves for f_2's trajectories in order to compare them against the EIB unicastFFA. For brevity, we restrict the depiction to f_2^A's cross-

traffic arrival bounding (mcastFFA step 1, Figure 10.4a) and f_2^B's delay bounding (mcastFFA step 2, Figure 10.4b). These derivations depict the crucial improvement of mcastFFA's proceedings in both of the analysis steps. They point out the reduction of the network and the increased tandem lengths.

mcastFFA step 1

We consider f_2^A's cross-traffic arrival bounding. Backtracking will be "local" to a single trajectory of a multicast cross-flow. In our example, we finally have established the possibility to apply the PMOO-principle when computing f_1^A's aggregate arrival bound aggrAB at server s_4 [8]. See $\alpha_4^{f_1^A}$ in the following left-over service curve derivation we require to bound cross-traffic arrivals:

$$\begin{aligned}
\beta^{\text{l.o.}f_2^A} &= \beta^{\text{l.o.}f_2^A}_{\langle 5,4\rangle} \qquad \text{(only single-hop interference so cutting is fine)}\\
&= \beta_5^{\text{l.o.}f_2^A} \otimes \beta_4^{\text{l.o.}f_2^A} = \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_4 \ominus \alpha_4^{f_1^A}\right)\\
&= \left(\beta_5 \ominus \left(\alpha^{f_1} \oslash \beta^{\text{l.o.}f_1}_{\langle 1,2\rangle}\right)\right) \otimes \left(\beta_4 \ominus \left(\alpha^{f_1} \oslash \beta^{\text{l.o.}f_1^A}_{\langle 1,2,3\rangle}\right)\right)
\end{aligned}$$

A cut of $\beta^{\text{l.o.}f_1^A}_{\langle 1,2,3\rangle}$ into $\beta^{\text{l.o.}f_1}_{\langle 1,2\rangle} \otimes \beta_3^{\text{l.o.}f_1^A}$ was needed in the EIB analysis, meaning that PMOO could not be implemented.

This advantage is also depicted in Figure 10.4a where f_1 retains its multicast shape in the mcastFFA's point of view.

mcastFFA step 2

For the second trajectory of f_2, f_2^B, our mcastFFA derives $\beta^{\text{l.o.}f_2^B} = \beta^{\text{l.o.}f_2^B}_{\langle 5,6\rangle}$. Again, we are not enforced to cut this trajectory's path (see Figure 10.4b) and in contrast to EIB we can apply alternative tandem analyses:

$$\begin{aligned}
\text{PBOO:} \quad \beta^{\text{l.o.}f_2^B} &= \beta^{\text{l.o.}f_2^B}_{\langle 5,6\rangle} \qquad \text{(cut enforced by SFA, no single-tandem analysis)}\\
&= \beta_5^{\text{l.o.}f_2^B} \otimes \beta_6^{\text{l.o.}f_2^B} = \left(\beta_5 \ominus \alpha_5^{f_1^B}\right) \otimes \left(\beta_6 \ominus \alpha_6^{f_1^B}\right)\\
&= \left(\beta_5 \ominus \left(\alpha^{f_1} \oslash \beta^{\text{l.o.}f_1}_{\langle 1,2\rangle}\right)\right) \otimes \left(\beta_6 \ominus \left(\alpha^{f_1} \oslash \beta^{\text{l.o.}f_1^B}_{\langle 1,2,5\rangle}\right)\right)
\end{aligned}$$

Note, that the actual trajectory of the cross-flow, f_1 or f_1^B, was automatically chosen correctly by the backtracking. Moreover, note the contrast to EIB: We can derive f_1^B's arrivals at s_6 with an end-to-end left-over service curve that, in turn, can make use of aggrAB.

$$\text{PMOO:} \quad \beta^{\text{l.o.}f_2^B} = \beta^{\text{l.o.}f_2^B}_{\langle 5,6 \rangle} \quad \text{(there is no enforced cut)}$$
$$= (\beta_5 \otimes \beta_6) \ominus \alpha_5^{f_1} = (\beta_5 \otimes \beta_6) \ominus \left(\alpha^{f_1} \oslash \beta^{\text{l.o.}f_1}_{\langle 1,2 \rangle} \right)$$

where $\beta^{\text{l.o.}f_1}_{\langle 1,2 \rangle}$ can be computed either by applying the left-over service curve derivation of SFA/PBOO or PMOO. This derivation is illustrated in Figure 10.4b.

Theoretical Evaluation

We conclude this section by a theoretical evaluation of mcastFFA against the related DNC approaches:

- *Relation to unicastFFA (Section 10.2.1.2):* The mcastFFA is a generalization of the unicastFFA. Analysis of unicast flows in either of the two steps remains unaffected (see f_3 in the running example).
- *Relation to unicastFFA transformation (Section 10.3):* Like the unicastFFA transformation, the mcastFFA is able to derive a PMOO end-to-end left-over service curve. However, it does so without the additional cross-traffic assumptions introduced by the unicastFFA transformation. I.e., there are less cross-flows to consider in the analysis, left-over service curves will be larger and delay bounds will be smaller. Thus, mcastFFA outperforms unicastFFA transformation.
- *Relation to EIB unicastFFA:* In comparison to EIB, we gained the ability to operate on end-to-end tandems. This constitutes increased flexibility to cut this tandem during the analysis: Our mcastFFA is compatible with SFA/PBOO, PMOO, aggrAB, or [5] for best attainable left-over service curves. This best solution to cut a tandem and combine sub-tandem results might coincide with EIB's enforced alternative, i.e., mcastFFA is indeed a generalization of EIB unicastFFA.

Before evaluating our contributions, let us briefly clarify their impact on the server backlog bound Q presented in Theorem 1. Deriving these bounds requires the arrival bounds of all flows at a server. I.e., in the DNC analysis procedures, (EIB) unicastFFA and mcastFFA, step 1 is crucial for the result accuracy; step 2 is not required. As shown with the running example, we improved the cross-traffic arrival bounding in case there are multicast flows present. Thus, backlog bounds are also improved by our contribution.

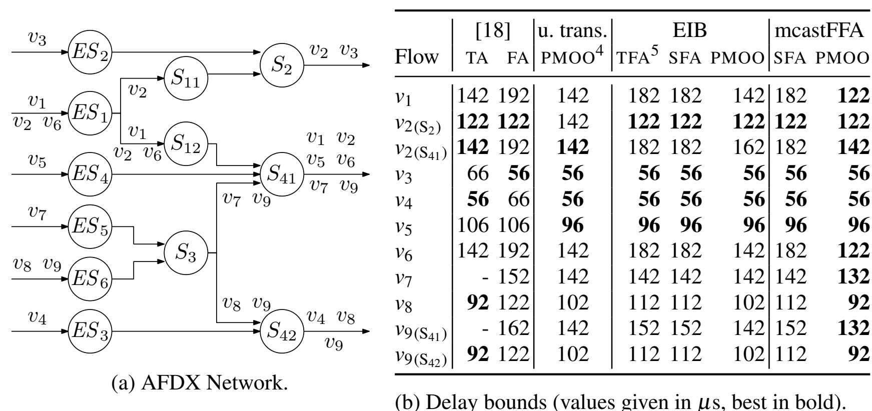
10.5 Numerical evaluation

In our numerical evaluation, we investigate achieved gains in terms of accuracy of end-to-end delay bounds. To that end, we provide two different comparisons. First, we benchmark our multicast feed-forward analysis (mcastFFA) against the related

approaches presented in Section 10.3. For our second set of results, a larger network evaluation, we implemented EIB and mcastFFA in the DiscoDNC tool [6].

10.5.1 Comparison to (Non-)Network Calculus Approaches

We study the AFDX network presented in [18]. This allows us to benchmark our proposed approach against the TA and FA since their numerical results are given in the literature. We note that we benchmark against the numerical results of TA and FA without the grouping properties extension since established DNC analyses do not yet take this property into account by default. The grouping property accounts for serialization of packetized flows when crossing links. We leave its implementation in the generalized DNC solutions, potentially based on [16, 13], to future work and restrict our comparison to the non-serialized results. Also, we use a fluid model for our evaluations. To achieve the best comparison possible with the related work on TA and FA, we also model store-and-forward behavior. This is achieved by an additional latency at every server that delays packet forwarding by the time required for full reception of a package of maximum size, $\max(pkg_size)/R$. Using their parameters defining service and arrivals, this enables us to confirm the DNC delay bounds[5] given in [18].



(a) AFDX Network.

Flow	[18] TA	[18] FA	u. trans. PMOO[4]	EIB TFA[5]	EIB SFA	EIB PMOO	mcastFFA SFA	mcastFFA PMOO
v_1	142	192	142	182	182	142	182	**122**
$v_{2(S_2)}$	**122**	**122**	142	**122**	**122**	**122**	**122**	**122**
$v_{2(S_{41})}$	**142**	192	**142**	182	182	162	182	**142**
v_3	66	**56**	**56**	**56**	**56**	**56**	**56**	**56**
v_4	**56**	66	**56**	**56**	**56**	**56**	**56**	**56**
v_5	106	106	**96**	**96**	**96**	**96**	**96**	**96**
v_6	142	192	142	182	182	142	182	**122**
v_7	-	152	142	142	142	142	142	**132**
v_8	**92**	122	102	112	112	102	112	**92**
$v_{9(S_{41})}$	-	162	142	152	152	142	152	**132**
$v_{9(S_{42})}$	**92**	122	102	112	112	102	112	**92**

(b) Delay bounds (values given in μs, best in bold).

Fig. 10.5: AFDX network evaluation of [18], extended with DNC's EIB and mcast-FFA.

Our comparison focuses on mcastFFA with PMOO, TA and FA delay bounds. We observe (see Figure 10.5) that mcastFFA at least matches the bounds of the other

[4] unicastFFA transformation approach with the stated PMOO end-to-end left-over service curve derivation.

[5] Remember, that EIB with TFA corresponds to the multicast TFA analysis presented as related work in Section 10.3.

methods compared here. A maximum gain of 5.86% compared to TA and 18.58% compared to FA is achieved in this small AFDX scenario.

Key observations w.r.t. the performance of DNC analyses confirm our theory:

- mcastFFA with PMOO shows expected gains compared to the multicast TFA[5].
- EIB with PMOO does not make full use of the PMOO principle end-to-end on trajectories and thus is outperformed by mcastFFA with PMOO in most cases.
- For some trajectories of multicast flows, even TFA results are equal. Then, flow interference is non-existent. These cases do not occur in realistic networks.

10.5.2 An Industry-scale AFDX Data Network

In order to evaluate our method on a realistic use-case found in industrial applications, we evaluate an AFDX data network. We aim to confirm our hypothesis that the mcastFFA will have a more pronounced advantage over other approaches[6] in larger networks. To that end, we implemented EIB and mcastFFA in the DiscoDNC tool. We also extended the DiscoDNC by an AFDX topology generator following recommendation from [12] and with network parameters according to an Airbus A350 presented in [17]. This also allows us to provide the entire range of DNC analysis configurations pairing EIB or mcastFFA with TFA as well as SFA/PBOO or PMOO $\beta^{\text{l.o.}}$ computations. For brevity of presentation, we focus on the most relevant of these combinations, EIB with SFA, EIB with PMOO and mcastFFA with PMOO. All results were computed using aggrAB arrival bounding [8]. Note, that this is not a restriction. Using segregated arrival bounding [11] or tandem matching arrival bounding [5] or any combination thereof is possible as well.

The network we generated according to these size parameters resulted in 650 multicast flows with 1112 trajectories in total. In order to compare the gains of mcastFFA PMOO against the other EIB methods, we used the relative difference, namely: $(d^{EIB} - d_{PMOO}^{mcastFFA})/d^{EIB}$. The empirical cumulative distribution (ECDF) over the studied A350-like network is illustrated on Figure 10.6. Key observations are unaffected: the mcastFFA procedure derives more accurate bounds than EIB. On average, mcastFFA PMOO produces a reduction of 8% of the bound compared to EIB SFA and 6% compared to EIB PMOO. We also observed reduction of up to 25% for some flows. These observations confirm our hypothesis that mcastFFA's potential advantage over other DNC approaches increases with the network size.

We also observe EIB SFA delay bounds that undercut the mcastFFA PMOO (see positive ECDF values for the negative x-axis in Figure 10.6). The situation stems from a well-known phenomenon that allows SFA to theoretically outperform PMOO by an arbitrarily large margin [22]. However, the mcastFFA can be paired with any tandem analysis able to compute output bounds. Doing so with the Tandem Matching Analysis (TMA) proposed in [5] creates a single-best algebraic analysis for arbitrary multiplexing.

[6] Due to a lack of software tools, TA and FA are not included.

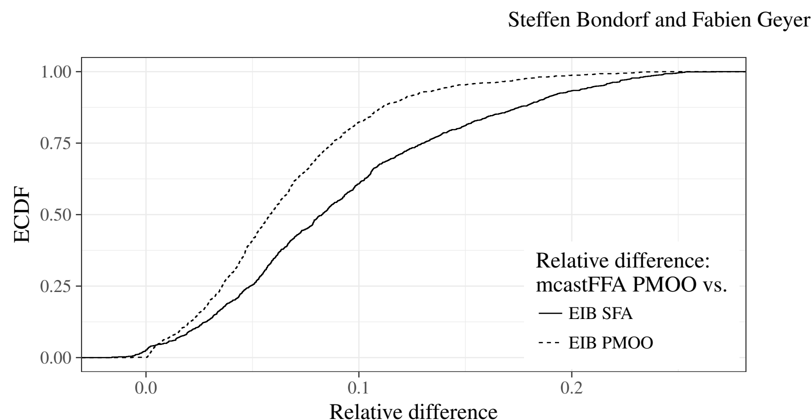


Fig. 10.6: ECDF of the relative difference between mcastFFA and the EIB methods

10.6 Conclusion and Outlook

In this chapter, we tackled the problem of analyzing multicast flows with deterministic network calculus. DNC was previously tailored to the analysis of unicast flows – a property which was assumed to invariantly hold. Therefore, previous approaches for the DNC analysis of multicast flows tried to adjust to this restriction by, e.g., pessimistic re-modeling of the network. This leads to inaccurate performance bounds and the development of alternative, non-DNC analyses to derive multicast flow guarantees. In contrast, we generalized DNC unicast feed-forward analysis to a multicast one.

In theoretical and numerical evaluations we showed that our contribution results in a single best DNC analysis for multicast flows, the mcastFFA with PMOO. Not only does it outperform any other DNC approach, the evaluation of an AFDX scenario from the literature also shows that DNC achieves at least the results of competitors (Trajectory Approach and Forward Analysis), even outperforming them in a considerable amount of cases.

Existing AFDX networks as deployed in existing Airbus aircraft such as the A380 are larger and more complex than the ones presented in this evaluation [25]. They consist of $\sim$1000 multicast flows (virtual links, VLs) that have an average of $\sim$6.5 trajectories per VL [1]. Therefore, the improvements we achieve with DNC's PMOO in conjunction with mcastFFA is expected to be even larger in practice.

Moreover, the presented mcastFFA has the flexibility to be combined with any DNC tandem analysis and improvement thereof. For instance, [9], [5], FIFO multiplexing service analysis [3] or packetization [13] can tighten guarantees and restriction to finite domains can accelerate the analysis [19].

Acknowledgments The authors would like to thank Bruno Oliveira Cattelan for his work on implementing the explicit intermediate bounds analysis and the multicast feed-forward analysis in the Disco Deterministic Network Calculator.

References

1. Bauer, H.: Analyse pire cas de flux hétérogènes dans un réseau embarqué avion. Ph.D. thesis, Université de Toulouse (2011)
2. Bauer, H., Scharbarg, J., Fraboul, C.: Applying and Optimizing Trajectory approach for performance evaluation of AFDX avionics network. In: Proc. of IEEE ETFA (2009)
3. Bisti, L., Lenzini, L., Mingozzi, E., Stea, G.: Numerical Analysis of Worst-Case End-to-End Delay Bounds in FIFO Tandem Networks. Springer Real-Time Systems Journal (2012)
4. Bondorf, S., Geyer, F.: Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows. In: Proc. of EAI ValueTools (2016)
5. Bondorf, S., Nikolaus, P., Schmitt, J.B.: Quality and Cost of Deterministic Network Calculus – Design and Evaluation of an Accurate and Fast Analysis. In: Proc. of ACM SIGMETRICS (2017)
6. Bondorf, S., Schmitt, J.B.: The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus. In: Proc. of EAI ValueTools (2014)
7. Bondorf, S., Schmitt, J.B.: Boosting Sensor Network Calculus by Thoroughly Bounding Cross-Traffic. In: Proc. IEEE INFOCOM (2015)
8. Bondorf, S., Schmitt, J.B.: Calculating Accurate End-to-End Delay Bounds – You Better Know Your Cross-Traffic. In: Proc. of EAI ValueTools (2015)
9. Bondorf, S., Schmitt, J.B.: Improving Cross-Traffic Bounds in Feed-Forward Networks – There is a Job for Everyone. In: Proc. of GI/ITG MMB & DFT (2016)
10. Le Boudec, J.-Y., Thiran, P.: Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Springer (2001)
11. Bouillard, A.: *Algorithms and Efficiency of Network Calculus*. Habilitation thesis, ENS (2014)
12. Boyer, M., Navet, N., Fumey, M.: Experimental Assessment of Timing Verification Techniques for AFDX. In: Proc. of ERTS (2012)
13. Boyer, M., Roux, P.: A common framework embedding network calculus and event stream theory. In: Proc. of IEEE ETFA (2016)
14. Chang, C.S.: Performance Guarantees in Communication Networks. Springer (2000)
15. Geyer, F., Carle, G.: Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches. IEEE Communications Magazine (2016)
16. Grieu, J.: Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques. Ph.D. thesis, Institut National Polytechnique de Toulouse (2004)
17. Hotescu, O., Jaffres-Runser, K., Scharbarg, J.L., Fraboul, C.: Towards quality of service provision with avionics full duplex switching. In: Euromicro ECRTS, Work-in-Progress Session (2017)
18. Kemayo, G., Benammar, N., Ridouard, F., Bauer, H., Richard, P.: Improving AFDX End-to-End delays analysis. In: Proc. of IEEE ETFA (2015)
19. Lampka, K., Bondorf, S., Schmitt, J.B., Guan, N., Yi, W.: Generalized finitary Real-Time calculus. In: Proc. IEEE INFOCOM (2017)
20. Martin, S., Minet, P.: Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class. In: Proc. of IPDPS (2006)
21. Migge, J.: L'ordonnancement sous contraintes temps-réel un modèle à base de trajectoires. Ph.D. thesis, INRIA Sophia Antipolis (1999)
22. Schmitt, J.B., Zdarsky, F.A., Fidler, M.: Delay Bounds under Arbitrary Multiplexing: When Network Calculus Leaves You in the Lurch. . . . In: Proc. of IEEE INFOCOM (2008)
23. Schmitt, J.B., Zdarsky, F.A., Martinovic, I.: Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once. In: Proc. of GI/ITG MMB (2008)
24. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing and Microprogramming (1994)
25. Tobeck, N.: Enforcing Domain Segregation in Unified Cabin Data Networks. In: Proc. of IEEE/AIAA DASC (2017)

A.1.3 Virtual Cross-Flow Detouring in the Deterministic Network Calculus Analysis

This work was published in *IFIP Networking 2020*, 2020 [21].

Virtual Cross-Flow Detouring in the Deterministic Network Calculus Analysis

Steffen Bondorf
Faculty of Mathematics, Center of Computer Science
Ruhr University Bochum, Germany

Fabien Geyer
Technical University of Munich | Airbus CRT
Munich, Germany

***Abstract*—Deterministic Network Calculus (DNC) is commonly used to compute bounds on the worst-case communication delay in data networks. It provides various analyses to derive these bounds from a model, giving different tradeoffs between accuracy and analysis efficiency. Improving the tradeoff led to increasingly complex algorithms. We set out to design a novel DNC algorithm that is of low complexity while still providing competitive delay bounds. To achieve this goal, we make use of the insight that added pessimism in the model can alleviate more severe limitations of the DNC analysis. To that end, we introduce the concept of *virtual cross-flow detouring* where data flows are assumed to cross additional servers. Ultimately, we provide a heuristic that is simple, fast and high-quality. We show in numerical evaluations that our detouring not only provides a competitive alternative, it also outperforms current algebraic algorithms' delay bounds for >50% of analyzed flows.**

I. Introduction

The correctness of many safety-critical applications is based upon formally verifying upper bounds on the end-to-end delay of data communication. They ensure proper functionality of the system. Deterministic Network Calculus (DNC) is a mathematical framework that has been applied for this verification in a wide variety of applications such as virtual machine placement in data-centers [1], in aerospace for the certification of fly-by-wire avionics [2], or admission control in self-modeling sensor networks [3]. To achieve valid bounds in the DNC framework, the analysis computing them adds some pessimism along the way. This may lead to over-provisioning of network resources and should thus be minimized. There have been efforts to extend the DNC with new results accordingly [4, 5, 6, 7] as well as efforts to recombine existing results to mitigate addition of pessimism [8, 6, 7]. Both of these streams of improvements have resulted in ever more complex algorithms; the tradeoff between complexity and quality became an active research topic in DNC. E.g., this tradeoff was improved by technical advances [9, 10], using machine learning in heuristics [11, 12], or recombination with other known results [13, 14]. In this paper, we will present a novel result called *virtual cross-flow detouring*, in short *detouring*. We integrate detouring into DNC to create a low-complexity analysis algorithm. The result of this integration is not only a considerably less complex and faster to execute heuristic. Its delay bounds also achieve a high level of quality, even exceeding those of the currently best fast analysis algorithm in the majority of our samples.

ISBN 978-3-903176-28-7 ©2020 IFIP

The DNC framework consists of two parts: modeling and analysis. A minimal DNC model provides the network topology and functions that either bound resource availability or demand at queueing locations (service curves and arrival curves). A DNC analysis has the objective to derive a bound on a specific flow's end-to-end delay when crossing the modeled network. The complexity in computing accurate delay bounds arises from the following characteristics of a minimal model:

- Arrival curves are provided per-flow at network entrance.
- Service curves bound the aggregate forwarding capability.

Yet, the DNC analysis aims to compute per-flow delay bounds.

In this paper, we propose virtual cross-flow detouring as an addition to existing analyses. Detouring defines a new degree of freedom in the search for the best tradeoff between length of analyzed server sequences (tandems) and flow aggregation. The main idea is that, if a cross-flow is detoured over (parts of) another flow's path, both can be treated by the analysis algorithm as an aggregate on a longer tandem. Despite the additional load at servers to be detoured over, this approach attains improved, valid delay bounds under certain conditions.

The addition of pessimism is not an entirely novel idea, a proof of concept that is considerably more restrictive than our detouring was presented in [15, 16]. This work proposed to prolong flows over the end of their respective paths, not to add servers at any location. The proposed exhaustive prolongation algorithm has two main characteristics: it is nearly infeasible to execute and the improvements to delay bounds are negligible. Secondly, focusing on longer tandems was also attempted in the recent literature [7]. While this can indeed lead to improved delay bounds, it becomes forbiddingly complex and infeasible to execute, too. We provide the first algorithm that makes ideas from both these concepts feasible to execute, even in combination, and without the use of techniques that do not allow for traceability of the solution process like optimization [5, 17] or machine learning [18, 11]. Despite the necessary measures to reduce computational complexity of the proposed algorithm, we observed that more than 50 % of delay bounds improved in every single network we analyzed (taken from [6]). Execution times increased by 42.6 % when adding detouring to a known DNC analysis, yet, creating a heuristic that is vastly faster than other similarly accurate analyses.

© IFIP, 2020. This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution. The definitive version will be published in IFIP Networking 2020.

The paper is structured as follows: Section II presents the DNC background. In Section III, we provide the virtual cross-flow detouring idea and a heuristic. Section IV benchmarks against existing analyses before Section V draws conclusions.

II. Deterministic Network Calculus Background

An extensive treatment of DNC can be found in [19, 20]. For brevity, we only presents the required background to understand virtual cross-flow detouring. DNC builds non-negative, wide-sense increasing functions that are used to lower bound resource availability guarantees (service curve β) or upper bound demand (arrival curve α), both in interval time. We abbreviate affine arrival curves (so-called token buckets) as $\alpha = \gamma_{r,b}(t) = \{rt + b\} \cdot \mathbb{1}_{t>0}$ and affine service curves (so-called rate latencies) as $\beta = \beta_{R,T}(t) = R \cdot \max\{0, T - t\}$.

We assume three further properties that are not explicitly modeled by these curves:

- order of data within a flow will not change (FIFO per flow),
- no knowledge about flow multiplexing in a server's queue is present (blind multiplexing of flows),
- multiplexing with cross-flows impacts the analyzed flow once per shared path (Pay Multiplexing Only Once, PMOO).

Hence, it is beneficial to analyze a flow over a long sequence of servers to capture the effect of the PMOO property. The work of [4] proposed an analysis implementing the PMOO property under the first two assumptions, known as PMOO Analysis (PMOOA). In this work, we extensively apply the following computation. It lower bounds the minimum residual service on a sequence of servers.

Theorem 1: The affine PMOOA left-over service curve $\beta^{\text{l.o.}} = \beta_{R^{\text{l.o.}},T^{\text{l.o.}}}$ for an analyzed flow of interest (foi) on a tandem of servers $\mathcal{T}$ is computed as

$$R^{\text{l.o.}} = \bigwedge_{s\in\mathcal{T}} \left(R_s - \sum_{(f\in s)\setminus\text{foi}} r^f \right)$$

$$T^{\text{l.o.}} = \frac{\sum_{(f\in\mathcal{T})\setminus\text{foi}} b^f + \sum_{s\in\mathcal{T}} \left(T_s \cdot \sum_{(f\in s)\setminus\text{foi}} r^f\right)}{R^{\text{l.o.}}} + \sum_{s\in\mathcal{T}} T_s$$

where $\wedge$ is the minimum, $s \in \mathcal{T}$ is a server on tandem $\mathcal{T}$, $f \in \mathcal{T}$ is a flow on $\mathcal{T}$, and $f \in s$ is a flow at server s.

For the computation of a single server's $\beta^{\text{l.o.}}$, we usually abbreviate the computation with the binary operator $\ominus$ to $\beta\ominus\alpha$. The two major known issues of Theorem 1 are:

1) Cross-flow bursts are served with the foi path's minimum left-over service rate $R^{\text{l.o.}}$, i.e., with the minimum across the entire tandem. Thus, $\beta^{\text{l.o.}}$ cannot benefit from increased service curves for individual servers on the tandem [5].

2) Arrival curves are required per set of cross-flows sharing one subpath of the foi path. This is known as segregation [7].

A recent, very accurate analysis called Tandem Matching (TMA, [6]) proposed an exhaustive search among all possible tradeoffs between these two aspects. Thus, TMA applies PMOOA left-over service computations. Large analysis complexity stems from finding the best tandems (tandem matching), not Theorem 1. Recently, it was proposed to replace the exhaustive search with machine learning predictions [11].

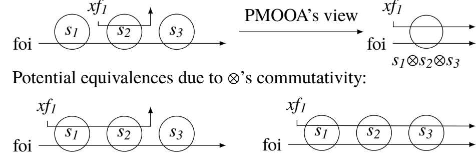


Figure 1: Potentially equivalent networks for a PMOOA due to commutativity of the DNC convolution $\otimes$.

The intuition behind PMOOA is simple: convolve the tandem of servers into a single system before subtracting the interfering arrivals. While this implements the PMOO principle, the underlying convolution causes issue 1. Convolution is commutative and therefore multiplexing cannot be localized to exactly the server where it occurs. In this paper, we look at PMOOA's issue 1 from a different angle: In Figure 1 we might be able to detour cross-flow xf_1 over server s_1 or s_3 without incurring a penalty for this added pessimism of distributing server resources among more flows. The conditions for such a PMOOA-equivalence (four equal $\beta^{\text{l.o.}}$ in Figure 1 by Theorem 1) of the tandems are: all curves are affine with $T_{s1} = T_{s3} = 0$, $R_{s_2} \geq R_{s_1}$, $R_{s_2} \geq R_{s_3}$.

III. Virtual Cross-Flow Detouring

Detouring is a simple extension of the DNC analysis that adds servers to cross-flows' paths (see Figure 2). Similar to improved speed of existing servers (issue 1), adding servers cannot improve the result of Theorem 1. Yet, it can have a positive impact by allowing the analysis to derive better network-internal arrival curves – output bounds derived as $\alpha \oslash \beta$ where $\oslash$ is the min-plus deconvolution [19]. We use output bounds to get the arrivals of cross-flows at a location where these flows interfere with a different, analyzed flow.

A. Detouring at the Front

In Figure 2, we are interested in a bound on the output after the 2-server tandems. We aggregate flows as much as possible and with $\alpha \oslash \beta_x \oslash \beta_y = \alpha \oslash (\beta_x \otimes \beta_y)$ ([19] Thm 3.1.12), the computation can progress server by server without losing the PMOO-benefits when convolving their service curves first. In Figure 2a, the output of server s_1 is bounded by

$$\alpha_{s_1}{}' = \gamma_{r^{f_1},b^{f_1}} \oslash \beta_{R_{s_1},T_{s_1}} = \gamma_{r^{f_1},b^{f_1}+r^{f_1}T_{s_1}}$$

and the output bound of server s_2, the final result, is

$$(\alpha_{s_1}{}' + \alpha_{s_2}) \oslash \beta_{s_2} = \gamma_{r^{f_1}+r^{f_2},b^{f_1}+b^{f_2}+r^{f_1}(T_{s_1}+T_{s_2})+r^{f_2}T_{s_2}}.$$

For this flow detouring at the front (Det$_{\text{front}}$) version in Figure 2b, we assume both flows already multiplexed in

Figure 2: Adding servers at the front of a flow's path.

s_1's queue and therefore we compute the output bound for the aggregate of both flows. In general, aggregation benefits the DNC analysis for this reason, yet, as we will see, flow detouring at the front introduces pessimism that outweighs the aggregation benefits in the output bound computation, too:

$$\begin{aligned}
\alpha_{s_1}^{\mathrm{Det_{front}}\,\prime} &= (\gamma_{r^{f_1},b^{f_1}} + \gamma_{r^{f_2},b^{f_2}}) \oslash \beta_{R_{s_1},T_{s_1}} \\
&= (\gamma_{r^{f_1},b^{f_1}} \oslash \beta_{R_{s_1},T_{s_1}}) + (\gamma_{r^{f_2},b^{f_2}} \oslash \beta_{R_{s_1},T_{s_1}}) \\
&= \gamma_{r^{f_1},b^{f_1}+r^{f_1}T_{s_1}} + \gamma_{r^{f_2},b^{f_2}+r^{f_2}T_{s_1}}
\end{aligned}$$

Note, that we applied distributivity of $\oslash$ over $+$ for rate-latency service and token-bucket arrivals [3]. This is not necessary at this point but it results in two separate output bound computations, nicely illustrating the increase of f_2's maximum burstiness arriving at s_2 by $r^{f_2}T_1$. The disadvantage is carried over to the output of server s_2 of Figure 2b:

$$\begin{aligned}
\alpha_{s_2}^{\mathrm{Det_{front}}\,\prime} &= \alpha_{s_1}^{\mathrm{Det_{front}}\,\prime} \oslash \beta_{s_2} \\
&= (\gamma_{r^{f_1},b^{f_1}+r^{f_1}T_{s_1}} + \gamma_{r^{f_2},b^{f_2}+r^{f_2}T_{s_1}}) \oslash \beta_{R_{s_2},T_{s_2}} \\
&= \gamma_{r^{f_1},b^{f_1}+r^{f_1}(T_{s_1}+T_{s_2})} + \gamma_{r^{f_2},b^{f_2}+r^{f_2}(T_{s_1}+T_{s_2})}
\end{aligned}$$

We have seen that flow detouring at the front will result in worse bounds – even if flows can be aggregated for output bounding. Next we show how flow detouring at the front can lead to better bounds nonetheless.

B. Detouring of Cross-traffic: Improving Bounds Nonetheless

In this section, we demonstrate that detouring over a server added to the middle of a flow's path can improve delay bounds derived by DNC despite the problems illustrated in the underlying detouring at the front.

We investigate the network shown in Figure 3a [7]. The output of cross-flows xf_1 and xf_2 after server s_0 needs to be bounded. Current DNC alternatives at s_0 are

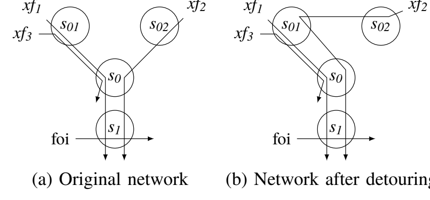
• *Maximize aggregation.* This enforces a hop-by-hop analysis due to the fork above s_0 and xf_1 cannot benefit from PMOOA when subtracting the impact of xf_3 on the tandem s_{01}, s_0.

• *Maximize tandem length.* This strategy allows for implementation of the PMOO principle, yet, it requires to segregate xf_1 and xf_2. I.e., these flows are analyzed individually and assume mutually exclusive worst-case system behavior at s_0.

With the PMOOA (Theorem 1) and virtual flow detouring in the analysis, we can benefit from the PMOO principle when subtracting cross-flow xf_3 and from aggregating xf_1 and xf_2 (see Figure 3b). Detouring is paid for by a different penalty (cf. $\mathrm{Det_{front}}$ in Section III-A), creating a new tradeoff that can beat the two existing strategies. The longer tandem will be able to hold more data in transit (added pessimism), and the issues of PMOOA prevent introduction of dangerous optimism by making it lack the ability to distribute load in a better way than on the original tandem. In summary, it is key to virtually detour a flow over its cross-flows such that the PMOOA has an impact. Then, the analysis of a more pessimistic setting can indeed result in better output bounds as we illustrate on Figure 3b next. For readability, we skip the s and f labels.

1) *Detouring* $\gamma^{\mathit{Detouring}} = \gamma_{r^{\mathit{Detouring}},b^{\mathit{Detouring}}}$ *arrivals at* s_1

$$\begin{aligned}
\gamma^{\mathrm{Detouring}} &= ((\alpha_2 \oslash \beta_{02}) + \alpha_1) \oslash ((\beta_{01} \otimes \beta_0) \ominus \alpha_3) \\
&= ((\gamma_{r_2,b_2} \oslash \beta_{R_{02},T_{02}}) + \gamma_{r_1,b_1}) \\
&\quad \oslash ((\beta_{R_{01},T_{01}} \otimes \beta_{R_0,T_0}) \ominus \gamma_{r_3,b_3}) \\
&= (\gamma_{r_2,b_2+r_2T_{02}} + \gamma_{r_1,b_1}) \oslash (\beta_{R_{01}\wedge R_0,T_{01}+T_0} \ominus \gamma_{r_3,b_3}) \\
&= \gamma_{r_1+r_2,b_1+b_2+r_2T_{02}} \oslash \beta_{(R_{01}\wedge R_0)-r_3,T_{01}+T_0+\frac{b_3+r_3(T_{01}+T_0)}{(R_{01}\wedge R_0)-r_3}}
\end{aligned}$$



(a) Original network (b) Network after detouring

Figure 3: Cross-flow detouring in the network [7] known to benefit from a longer tandem analysis of xf_1.

with $r^{\mathrm{Detouring}} = r_1 + r_2$ and

$$\begin{aligned}
b^{\mathrm{Detouring}} &= b_1 + b_2 + r_2T_{02} \\
&\quad + (r_1 + r_2)\left(T_{01} + T_0 + \frac{b_3 + r_3(T_{01} + T_0)}{(R_{01} \wedge R_0) - r_3}\right) \\
&= b_1 + b_2 + r_2T_{02} \\
&\quad + r_2T_{01} + (r_1 + r_2)T_0 + r_1T_{01} + (r_1 + r_2)\frac{b_3 + r_3(T_{01} + T_0)}{(R_{01} \wedge R_0) - r_3}
\end{aligned}$$

From the literature, we get two further valid bounds for the arrival of $xf_1 + xf_2$ at s_1, derived as follows:

2) *Aggregate Arrival Bounding's* γ^{AggrAB} *[8] in Figure 3a*

$$\begin{aligned}
\gamma^{\mathrm{AggrAB}} &= \gamma_{r_1+r_2,\, b_1+b_2+r_1T_{01}+r_2T_{02}} \\
&\qquad {}_{+r_1\frac{b_3+r_3T_{01}}{R_{01}-r_3}+(r_1+r_2)\left(T_0+\frac{b_3+r_3(T_{01}+T_0)}{R_0-r_3}\right)}
\end{aligned}$$

3) *Segregated Arrival Bounding's* γ^{SegrAB} *[7] in Figure 3a*

$$\begin{aligned}
\gamma^{\mathrm{SegrAB}} &= \gamma_{r_1+r_2,\, b_1+b_2+r_1T_{01}+r_2T_{02}+(r_1+r_2)T_0} \\
&\qquad {}_{+r_1\frac{b_2+b_3+r_3T_{01}+(r_2+r_3)T_0}{(R_{01}-r_3)\wedge(R_0-r_2-r_3)}+r_2\frac{b_1+b_3+(r_1+r_3)T_0}{R_{02}\wedge(R_0-r_1-r_3)}}
\end{aligned}$$

We see that all three alternatives compute the same arrival rate at s_1, $r_1 + r_2$, and that there are common terms in the arrival burstiness, $b_1 + b_2 + r_1T_{01} + r_2T_{02} + (r_1 + r_2)T_0$. We assume a network with homogeneous rates $R_0 = R_{01} = R_{02} =: R$, $r_1 = r_2 = r_3 =: r$ and compare the remaining burst terms.

• $\gamma^{\mathrm{Detouring}} < \gamma^{\mathrm{AggrAB}}$:

$$\begin{aligned}
& rT_{01} + 2r\frac{b_3 + r(T_{01} + T_0)}{R - r} < \\
& \qquad r\frac{b_3 + rT_{01}}{R - r} + 2r\frac{b_3 + r(T_{01} + T_0)}{R - r} \\
\Leftrightarrow \quad & T_{01}(R - 2r) < b_3
\end{aligned}$$

• $\gamma^{\mathrm{Detouring}} < \gamma^{\mathrm{SegrAB}}$:

$$\begin{aligned}
& rT_{01} + 2r\frac{b_3 + r(T_{01} + T_0)}{R - r} < \\
& \qquad r\frac{b_2 + b_3 + rT_{01} + 2rT_0}{R - 2r} + r\frac{b_1 + b_3 + 2rT_0}{R - 2r} \\
\Leftrightarrow \quad & (R - r)T_{01} - 2rT_0 < b_1 + b_2
\end{aligned}$$

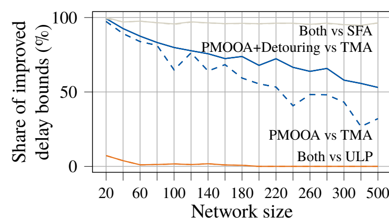


Figure 4: Delay bound improvements.

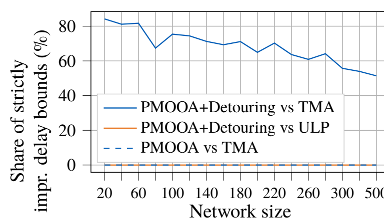


Figure 5: Strict delay bound reductions.

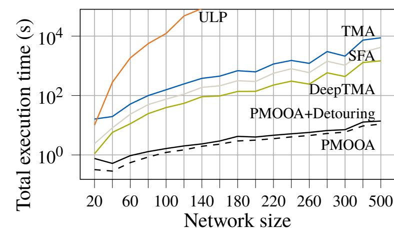


Figure 6: Execution times.

In [7], parameters were fixed to show $\gamma^{\text{SegrAB}} < \gamma^{\text{AggrAB}}$: latencies are 0, $b_1 = b_2 = 0$. With the former, virtual detouring will attain better results. Adding the latter, its results will equal the SegrAB. The remaining parameters are constant. In a homogeneous network modeled with affine curves, detouring can theoretically outperform the AggrAB and SegrAB output bounding strategies by an arbitrarily large margin.

C. Application to Large Feedforward Networks

In Figure 3, there is only a single sensible detouring alternative. In larger networks with flows taking different paths, there may be several virtual detouring alternatives in order to allow for the PMOO principle's application to different sets of their cross-flows – the amount of alternatives certainly increases with the network size. This creates potential for a combinatorial explosion. Therefore, we will provide a simple yet effective heuristic to select one single detouring path.

From TMA [6] we learn the reason for effort in the DNC analysis: The analysis is executed with recursive backtracking [8], starting from the foi and backtracking over cross-flows. Bounds on arrivals of these cross-flows are not known before the backtracking terminated and results for the piled up recursion levels are computed. During backtracking, these still missing results are yet required to find the best bound computation for a recursion level. Thus, the exhaustive search keeps all alternatives alive until all information is known.

From DeepTMA [11] we learn that we need a non-exhaustive heuristic. Our detouring heuristic only operates on the invariant data available at any location in the network: service curves and the sole presence of flows (without arrival curves / output bounds). The former is not of much use without arrival curves and thus, we use the latter. We interpret presence of flows as potential for cross-flow aggregation and additional servers to detoured over. The decision on detouring can be easily done server-by-server, embedded into the backtracking: *At any server, simply check all in-links and detour over the one that has the most flows on it*. If there are multiple alternatives, randomly choose one of them. Note, that the random choice we opted for in our heuristic negatively impacts reproducibility of our evaluation results. But any other tie-breaker would increase the complexity of the detouring heuristic. We terminate the detouring-path search when one of these conditions is met:
i) there is no cross-flow or analyzed flow left to prolong over
ii) bounds become infinite (long-term service < arrivals).

IV. Numerical Evaluation

A. Evaluation Setup

For benchmarks, we compute delay bounds in the networks from [6][1] (20 to 500 devices, 152 to 7504 flows). We mainly compare PMOOA with detouring (PMOOA+Detouring), the Tandem Matching Analysis (TMA) [6] paired with the Burst Cap (BC) [13] mechanism and the optimization-based ULP analysis [17]. The basis for our PMOOA+Detouring implementation is the open-source NetCal DNC v2.6 [21][2].

B. Improvement over Other Analyses

We compare in this section the resulting end-to-end delays produced by PMOOA+Detouring and compare them against TMA and SFA [19, 20]. We aim to derive the best delay bound for every flow. There are no semantics assigned; the flow with the best improvement could be the most important one.

We first evaluate how many flows have their end-to-end delay bound matched or reduced with the use of detouring, compared to the other analyses:

$$delay_{PMOOA+Detouring} \leq delay_X$$

Results are presented in Figure 4. PMOOA+Detouring is able to match or improve delay bounds of TMA for at least $53.0\,\%$ of the analyzed flows – this lowest value is obtained in the largest network. In contrast to PMOOA without Detouring, which matches at most $26.7\,\%$ of the analyzed flows compared in the largest network, the addition of detouring is beneficial. As expected, the use of detouring has thus almost no impact on the existing relation to SFA and ULP delay bounds [6].

As a second metric, we evaluate how many flows have their end-to-end delay bound strictly reduced, namely we use a strict inequality compared to before:

$$delay_{PMOOA+Detouring} < delay_X$$

Results are presented in Figure 5. As expected, PMOOA does not produce tighter bounds than TMA. Using detouring results in strictly tighter bounds for more than $51.4\,\%$ of the analyzed flows, meaning that our approach is able to outperform the tightness of TMA for more than half of the evaluated flows. Compared to ULP, no strict improvement is achieved, again an expected result for the same reasons as above.

[1] Available at: `https://github.com/NetCal/DNC_experiments`
[2] Available at: `https://github.com/NetCal/DNC`

C. Relative Change

We illustrated that our heuristic can match or even outperform PMOOA and TMA+BC. As PMOOA+Detouring competes with segregate arrival bounding of [7] (SegrPMOO), we benchmark against the results available for TMA+SegrPMOO. We compare delay bounds with the *relative change* metric:

$$RelativeChange_x = \left(delay_{Detouring} - delay_x\right) / delay_x$$

Results are presented in Figure 7 for the only two networks from the dataset that can be analyzed with TMA+SegrPMOO.

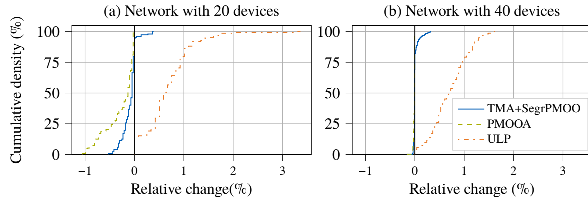


Figure 7: Relative change of PMOOA+Detouring compared to TMA+SegrPMOO, PMOOA and ULP.

As illustrated in Figure 7a, detouring can result in the reduction of end-to-end delay bounds of up to $1\,\%$ compared to PMOOA, or up to $0.5\,\%$ compared to TMA+SegrPMOO. This effect is less visible in Figure 7b, but detouring still results in $0.2\,\%$ in the best case.

D. Execution Time

In order to illustrate the impact of PMOOA+Detouring's small complexity, we compare the execution time of our heuristic with other analyses. We benchmark the different analyses by measuring the execution time per network, i.e. the overall elapsed wall clock time of an analysis for all the flows in a network size. Execution times presented here do not include the time to model a network or derive the server graph or adding the flows. These are not in focus of this work and they are the shared prerequisite for all the evaluated analyses.

Results are presented in Figure 6. All execution times were measured on the same machine with an Intel Xeon CPU E5420 at $2.50\,\text{GHz}$. For DeepTMA, no GPU acceleration was used. For ULP, CPLEX was used for solving the linear program.

PMOOA+Detouring takes advantage of the efficiency of PMOOA and outperforms all other analyses by at least two orders of magnitude, except pure PMOOA of course. On average, the addition of detouring (with a PMOOA on the detour) resulted in an increase of only $42.6\,\%$ in average, indicating that detours are rather short. Those results illustrate that the computational cost of using detouring is negligible compared to the gains in tightness illustrated earlier.

V. Conclusion

In this paper, we contribute *virtual cross-flow detouring* (detouring) to deterministic network calculus. We show that it is a powerful extension to the PMOOA analysis, resulting in delay bounds matching or even outperforming the state-of-the-art analyses that are considerably more involved. Our contribution is based on the counter-intuitive idea of adding pessimism to the model. Due to a previously frowned upon characteristic of PMOOA, we can compute better delay bounds nonetheless.

Our evaluation shows that detouring is able to outperform the previously best analyses, in particular TMA. Delay bounds of more than $50\,\%$ of the analyzed flows improved compared to TMA. This is achieved by only a small addition to PMOOA's execution time of $42.6\,\%$ on average. That makes our proposed heuristic not only comparable with TMA w.r.t. the computed delay bounds but also faster by two orders of magnitude.

References

[1] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. of ACM SIGCOMM*, 2015.

[2] F. Geyer and G. Carle, "Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches," *IEEE Communications Magazine*, vol. 54, no. 2, pp. 106–112, 2016.

[3] S. Bondorf and J. B. Schmitt, "Boosting sensor network calculus by thoroughly bounding cross-traffic," in *Proc. of IEEE INFOCOM*, 2015.

[4] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *Proc. of GI/ITG MMB*, 2008.

[5] J. B. Schmitt, F. A. Zdarsky, and M. Fidler, "Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch...," in *Proc. of IEEE INFOCOM*, 2008.

[6] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 1, no. 1, pp. 16:1–16:34, 2017.

[7] ——, "Catching corner cases in network calculus – flow segregation can improve accuracy," in *Proc. of GI/ITG MMB*, 2018.

[8] S. Bondorf and J. B. Schmitt, "Calculating accurate end-to-end delay bounds – you better know your cross-traffic," in *Proc. of EAI ValueTools*, 2015.

[9] N. Luangsomboon, R. Hesse, and J. Liebeherr, "Fast min-plus convolution and deconvolution on GPUs," in *Proc. of EAI ValueTools*, 2017.

[10] A. Scheffler, M. Fögen, and S. Bondorf, "The deterministic network calculus analysis: Reliability insights and performance improvements," in *Proc. of IEEE Intl. Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks (CAMAD)*, 2018.

[11] F. Geyer and S. Bondorf, "DeepTMA: Predicting effective contention models for network calculus using graph neural networks," in *Proc. of INFOCOM*, 2019.

[12] ——, "On the robustness of deep learning-predicted contention models for network calculus," 2019, arxiv:1911.10522.

[13] S. Bondorf and J. B. Schmitt, "Improving cross-traffic bounds in feed-forward networks – there is a job for everyone," in *Proc. of GI/ITG MMB & DFT*, 2016.

[14] A. Mifdaoui and T. Leydier, "Beyond the accuracy-complexity trade-offs of compositional analyses using network calculus for complex networks," in *Proc. of CRTS Workshop*, 2017.

[15] L. Bisti, L. Lenzini, E. Mingozzi, and G. Stea, "Estimating the worst-case delay in FIFO tandems using network calculus," in *Proc. of ICST ValueTools*, 2008.

[16] S. Bondorf, "Better bounds by worse assumptions – improving network calculus accuracy by adding pessimism to the network model," in *Proc. of IEEE ICC*, 2017.

[17] A. Bouillard, L. Jouhet, and É. Thierry, "Tight performance bounds in the worst-case analysis of feed-forward networks," in *Proc. of IEEE INFOCOM*, 2010.

[18] F. Geyer, "Performance evaluation of network topologies using graph-based deep learning," in *Proc. of EAI ValueTools*, 2017.

[19] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[20] A. Bouillard, M. Boyer, and E. Le Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*. Wiley, 2018.

[21] S. Bondorf and J. B. Schmitt, "The DiscoDNC v2 – a comprehensive tool for deterministic network calculus," in *Proc. of EAI ValueTools*, 2014.

A.1.4 The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds

This work was published in *Proceedings of the 2018 International Workshop on Network Calculus and Applications*, 2018 [69].

The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds

Fabien Geyer
Technical University of Munich (TUM)
D-85748 Garching b. München, Germany
Email: fgeyer@net.in.tum.de

Georg Carle
Technical University of Munich (TUM)
D-85748 Garching b. München, Germany
Email: carle@net.in.tum.de

***Abstract*—Deterministic network calculus offers a framework for providing guaranteed bounds on end-to-end delay and buffer usage in computer networks. Various network analysis methods have been proposed in order to reduce the impact of burstiness or multiplexing and provide tight performance bounds. Yet, the choice of which analysis method to use given a network to analyze is not straightforward as it has been shown in the literature that corner cases exist leading to poor tightness. We propose in this paper to take a new look at this question using insights from data and confirm that there is no clear winner when deciding which method to use. Based on those first results, we make the case for a network calculus heuristic in order to predict the bounds produced by a given network analysis method. Our main contribution is a heuristic based on graph-based deep learning, which is able to directly process networks of servers and flows. Via a numerical evaluation, we show that our proposed heuristic is able to accurately predict which analysis method will produce the tightest delay bound. We also demonstrate that the computational cost of our heuristic makes it of practical use, with average runtimes one or two order of magnitude faster than traditional analysis methods.**

I. Introduction

Performance guarantees are an essential part of network architecture and design in real-time networks such as safety critical systems [1]. In the case of large Ethernet networks, deterministic network calculus (DNC) [2] has been successfully used as a mathematical framework for validating and guaranteeing end-to-end delay requirements and buffer sizes. An important aspect of such framework is to achieve good tightness, namely minimizing the gap due to the pessimism of a method and the real worst-case. Various network analysis methods have been investigated in the literature to address this, by either focusing on specific effects such as multiplexing [3] or using methods based on optimization [4]. Yet, no clear guideline has been proposed with regards to choosing the appropriate method given a network to analyze.

To address this question, we propose in this paper to take a new look at this problem from the perspective of data. In a first step, we confirm previous results on corner cases of network analyses by evaluating them on randomly generated feed-forward networks. Based on those findings, we propose a heuristic for network calculus analyses using a neural network able to process graphs in order to predict the end-to-end latency bound of a given analysis method. Our heuristic is based on a transformation from the feed-forward server graph and the flows crossing it to a general graph which can then be analyzed using recent techniques from neural networks focused on graph analysis.

We demonstrate via a numerical evaluation that our trained neural network is able to predict end-to-end latency bounds with a relative median error of 2.5 %. While the predictions of the neural network cannot be directly used for giving performance guarantees, we show that such heuristic can be used for deciding which network analysis method to apply given a network and a flow of interest. Compared to a strategy of using only one analysis method, using our heuristic as a selection of which method to use leads to a relative reduction of the end-to-end delay bound of 12.79 % in average. Finally, we also evaluate the runtime of our heuristic and show that it is one order of magnitude faster than a standard network analysis method, highlighting its usability in practice.

The rest of this paper is organized as follows. We first give an overview of network calculus in Section II and investigate the state of the different network analysis methods using numerical evaluations. We present in Section III our heuristic based on graph-based deep learning. In Section IV, we numerically evaluate our approach and show an application of our heuristic. Related research studies are presented in Section V. Finally, Section VI concludes our work.

II. Deterministic network calculus

We present in this section a brief overview over deterministic network calculus and a numerical evaluation of its different network analysis methods. In DNC, a flow corresponds to unidirectional communications between a source and a destination, modeled as a function of its cumulative arrival of data. In order to compute bounds on flows, we are interested in the functions $A(t)$ corresponding to the data arriving in a given server s at time t, and $A'(t)$ the amount of data processed by the server at time t. Using this formalism, the following delay definition can then be derived:

Definition 1 (Flow delay): Assume a flow with input A and crosses a server s and results in the output A'. The (virtual) delay for a data unit arriving until time t is

$$D(t) = \inf\{\tau \geq 0 \mid A(t) \leq A'(t+\tau)\}$$

Instead of directly working with A, DNC makes use of the concept of arrival curves, which is a function bounding the maximal arrivals of a flow:

Definition 2 (Arrival curve): Given a flow with input A, a function α is an arrival curve for A iff

$$A(t) - A(s) \leq \alpha(t-s), \forall t, s, 0 \leq s \leq t$$

Definition 3 (Service curve): If the service provided by a server s for a given input A results in an output A', then s offers a service curve β iff

$$A'(t) \geq \inf_{0 \leq s \leq t} \{A(t-s) + \beta(s)\}, \forall t$$

A. (min, +) Algebra

DNC was formalized as a $(\min, +)$-algebraic framework in [5, 2], enabling an easier description of operations on flow and server descriptions. The $(\min, +)$ convolution and deconvolution of two functions f, g are defined as:

$$\text{Convolution: } (f \otimes g)(t) = \inf_{0 \leq s \leq t} \{f(t-s) + g(s)\}$$

$$\text{Deconvolution: } (f \oslash g)(t) = \sup_{s \geq 0} \{f(t+s) - g(s)\}$$

Using those $(\min, +)$ operations, one can rewrite the previous definitions as $A' \geq A \otimes \beta$ and $A \otimes \alpha \geq A$. Moreover, $(\min, +)$ convolution allows DNC to concatenate the service of consecutive servers $\langle 1, \ldots, n \rangle$ into a single service curve.

B. Network analysis methods

We review here the most common network analysis methods from DNC. We refer to [6] for a more in-depth review of the different methods proposed in the literature. We call the analyzed flow *flow of interest*, abbreviated here *foi*. The foi's path defines the sequence of servers that defines its end-to-end delay. Different methods have been proposed in the literature for bounding this end-to-end delay.

1) Total Flow Analysis (TFA) [2]: The TFA first computes per-server delay bounds. Each one holds for the sum of all the traffic arriving to a server. The flow's end-to-end delay bound is derived by summing up the individual server delay bounds on its path.

2) Separated Flow Analysis (SFA) [2]: The SFA is a direct application of other theorems: first compute the left-over service of each server on the foi's path, then concatenate them and finally derive the end-to-end delay bound. Deriving the end-to-end delay bound using only one service curve will consider the burst term of the foi only once, a property called Pay Burst Only Once (PBOO).

3) Pay Multiplexing Only Once (PMOO) [3]: The PMOO analysis first convolves the tandem of servers before subtracting the cross-traffic. Using this order, the bursts of the cross-traffic appear only a single time compared to the SFA analysis where the bursts are included at each server. Therefore, multiplexing with cross-traffic is only paid for once.

4) Arrival bounding methods: For more involved feed-forward networks, a procedure to combine tandem analyses with a network analysis have been proposed. [7] established two basic steps of the analysis: *1)* cross-traffic arrival bounding and *2)* flow of interest performance bounding. For the cross-traffic arrival bounding, the flows interfering with the foi are backtracked to their sources to derives the dependencies between the foi and its cross-flows in a recursive fashion. The PBOO and PMOO properties can then also be applied on the cross-traffic as shown in [8, 7].

C. Numerical comparison between methods

From the description of the different methods previously presented, the most promising method for producing a tight bound is PMOO. However, it was demonstrated in [9] that PMOO does not necessarily outperform SFA. These problems all aggravate in the analysis of entire networks, where accurately bounding cross-traffic is important.

In order to better understand the differences between the different network analysis methods previously described, we propose in this section to numerically investigate the bounds produced by each method on a dataset of 44 044 topologies[1]. Our methodology for evaluating the methods is as follows. We generated random feed-forward networks as illustrated in Figure 1, where up to 10 servers are connected in a daisy chain manner. For each server, a rate-latency curve is used with the rate and latency sampled from a uniform distribution. Up to 40 flows are then randomly generated with random sources and destination servers, with a token bucket arrival curve with the rate and burst sampled from a uniform distribution. Each topology is then evaluated using DiscoDNC [8] (version 2.4.0) using TFA, SFA and PMOO, with different arrival bounding methods. The arrival bounding methods are labeled as: *PMOO-AB* for `ArrivalBoundMethod.PMOO` in DiscoDNC, *PMOO-PF-AB* for `PER_FLOW_PMOO`, *PBOO-PF-AB* for `PBOO_CONCATENATION`, and *PBOO-PH-AB* for `PBOO_PER_HOP`. We refer to [8] for a complete explanations of those arrival bounding methods. The topologies are built such that they satisfy the feed-forward property, i.e., there are no cyclic dependencies between the flows. For our evaluation, we focus in this paper on end-to-end delay bounds.

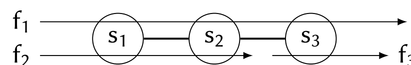


Figure 1: Example feed-forward network

The numerical results and comparison between analysis methods is presented in Figure 2. We first compare in Figure 2(a) and (b) the flows' end-to-end delay bounds against the best and worst delay bounds produced by the other methods. As expected, PMOO produces the tightest delay bounds for around 70 % of the analyzed flows, SFA for 54 %, and TFA for 1 %. Those numerical results are in line with [9, 7]. Surprisingly, we notice that PMOO produces the worst delay bounds in around 11 % of the studied flows, highlighting and confirming the existence of corner-cases. We also notice that the cross-traffic arrival bounding method has a noticeable influence on tightness.

Based on those results, we investigate in Figure 2(c) how much tightness is lost in case the network analysis does not

[1] Available here: `https://github.com/fabgeyer/dataset-itc30nc`

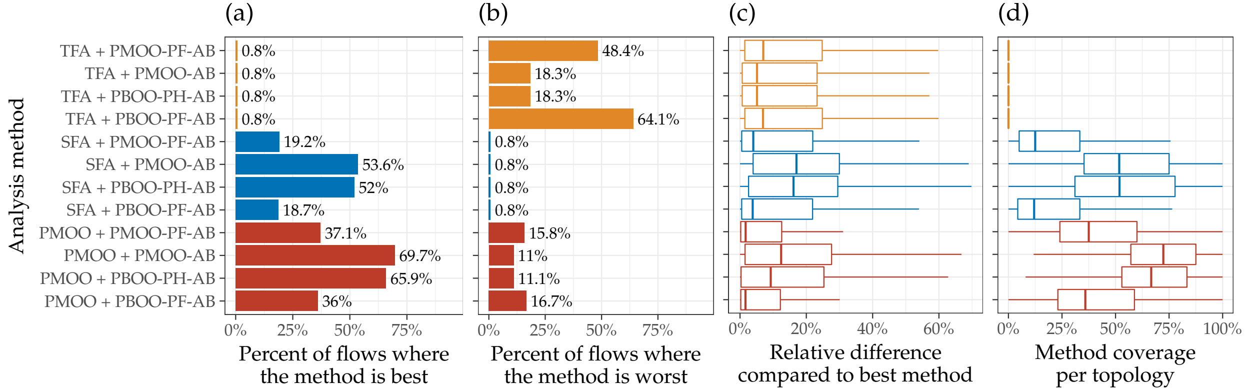


Figure 2: Evaluation and comparison of the different network analysis methods against different metrics. *(a)* Ratio of flows where a given method produces the tightest bound compared to the other methods, *(b)* respectively the worst bound. *(c)* Relative difference between delay bound of a given method and the best method when the given method does not provide the best bound. *(d)* Ratio of networks where a given method is able to produce the tightest bounds for all the flows of the topology.

produce the tightest bound. For each flow $\mathcal{F}$ where a given method M did not produce the tightest bound, we use the relative difference to numerically assess this lost tightness:

$$\mathit{Relative\ Difference}(\mathcal{F}, M) = \frac{D^M_{\mathcal{F}} - \min_m D^m_{\mathcal{F}}}{\min_m D^m_{\mathcal{F}}} \quad (1)$$

with $D^M_{\mathcal{F}}$ the end-to-end delay bound of flow $\mathcal{F}$ with analysis method M.

Finally, we evaluate the ability of a network analysis method to produce the tightest bounds for all flows in a given network topology. To numerically assess this notion, we define the *coverage per topology* for a given method as the ratio of number of flows where the method produced the tightest bounds divided by the total number of flows in the topology. Results are presented in Figure 2(d). Although the results are in line with the numerical results from Figure 2(a) and (b), we notice that given a network topology, using a single network analysis method will not produce the tightest bounds for all flows in the topology. This finding motivates an adaptive analysis for a given topology, where different network analysis methods would be used depending on the flow of interest.

III. DNC heuristic using Neural Networks

We introduce in this section a heuristic based on the concept of Graph Neural Network introduced in [10, 11]. The main intuition behind our approach is to map network topologies and flows to graphs. Those graph representations are then used as input for a neural network architecture able to process general graphs.

A. Presentation

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and target values of node v. The concept behind Graph Neural Networks is called *message passing*, where hidden representations of nodes are based on the hidden representations of their neighboring nodes. Those hidden representations are propagated through the graph using multiple iterations until a fixed point is found. The final hidden representation is then used for predicting a property about the node. This concept can be expressed as:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}_u^{(t-1)} \,\middle|\, u \in \textsc{Nbr}(v)\right\}\right) \quad (2)$$

$$\mathbf{o}_v = g\left(\mathbf{h}_v^{(t\to\infty)}\right) \quad (3)$$

$$\mathbf{h}_v^{(t=0)} = \mathit{init}\left(\mathbf{i}_v\right) \quad (4)$$

with $\mathbf{h}_v^{(t)}$ representing the hidden representation of node v at time t, $f(\cdot)$ a function which aggregates the different hidden representations, $\textsc{Nbr}(v)$ the set of neighboring nodes of v, $g(\cdot)$ a function for transforming the final hidden representation to the target values, and $\mathit{init}(\cdot)$ a function for initializing the hidden representations based on the input features.

The concrete implementation of the $f(\cdot)$ and $g(\cdot)$ functions are feed-forward neural networks with the special case that $f(\cdot)$ in Equation (2) is the sum of per-edge terms (as recommended by [11]) such that:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}^{(t-1)}_{\textsc{Nbr}(v)}\right\}\right) = \sum_{u \in \textsc{Nbr}(v)} f^*\left(\mathbf{h}_u^{(t-1)}\right) \quad (5)$$

with $f^*(\cdot)$ a feed-forward neural network. For $\mathit{init}(\cdot)$, the initial features are usually zero-padded to fit the dimensions of the hidden representations.

B. Extensions

We give in this section a brief overview of the extensions to GNNs which were used in this paper.

1) Gated Graph Neural Network: In order to improve the training of Graph Neural Networks, Li et al. [12] proposed Gated Graph Neural Networks (GG-NNs). This extension implements the function $f(\cdot)$ using a memory unit called Gated Recurrent Unit (GRU) [13] and unrolls Equation (2)

for a fixed number of iterations. The propagation of hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \tag{6}$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z\mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \tag{7}$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r\mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \tag{8}$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W}x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \tag{9}$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \tag{10}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function and $\odot$ the element-wise matrix multiplication. $\{\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}\}$ and $\{\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}\}$ are trainable weights matrices, and $\{\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}\}$ are trainable biases vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix. Equation (6) corresponds to the aggregation of messages from adjacent nodes as in Equation (5). Equations (7) to (10) correspond respectively to the reset gate, the update gate, the candidate output, and the output vector of a standard GRU cell [13].

2) Edge attention: A recent advance in neural networks has been the concept of *attention*, which provides the ability to a neural network to focus on a subset of its inputs. For the scope of GNNs, we introduce here so-called *edge attention*, namely we wish to give the ability to each node to focus only on a subset of its neighborhood. Formally, let $a^{(t)}_{(v,u)} \in [0,1]$ be the attention between node v and u. Equation (5) is then extended:

$$\mathbf{h}_v^{(t)} = \sum_{u\in\text{NBR}(v)} a^{(t)}_{(v,u)} \cdot f^*\left(\mathbf{h}_u^{(t-1)}\right) \tag{11}$$

$$\mathbf{a}^{(t)}_{(v,u)} = f_A\left(\mathbf{h}_v^{(t-1)}, \mathbf{h}_u^{(t-1)}\right) \tag{12}$$

C. Application to deterministic network calculus

In order to apply the concepts described in Sections III-A and III-B to network calculus analysis, we model the feed-forward server graph and the flows crossing it into graphs. Each servers is represented as a node in the graph, with edges corresponding to the connections between servers. Each flow is represented as a node with edges connecting it to the path of traversed servers. Since the order of the servers which is traversed by a flow plays a large influence in network calculus, so-called *path ordering* nodes are added on the edges between the flow node and the server nodes. Figure 3 illustrates this graph encoding with the network from Figure 1.

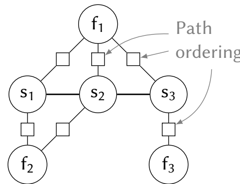


Figure 3: Graph encoding of the example topology. Square nodes represent additional nodes encoding path ordering.

Each node in the graph has the following input features:

- For server $\mathcal{S}$, we use the parameters of its rate-latency service curve: $\mathbf{i}_\mathcal{S} = [rate_\mathcal{S}, latency_\mathcal{S}]$;
- For flow $\mathcal{F}$, we use the parameters of its token bucket arrival curve: $\mathbf{i}_\mathcal{F} = [rate_\mathcal{F}, burst_\mathcal{F}]$;
- For a path ordering node $\mathcal{O}$, a categorical encoding of the hop index is used as input feature. We use standard one-hot encoding, namely $\mathbf{i}_\mathcal{O}$ is a vector with a one at the hop index, and zeros otherwise (e.g.: in Figure 3, we have $\mathbf{i}_\mathcal{O}^{f_1-s_1} = [1,0,0]$, $\mathbf{i}_\mathcal{O}^{f_1-s_2} = [0,1,0]$, $\mathbf{i}_\mathcal{O}^{f_1-s_3} = [0,0,1]$).

For the output prediction of each node representing a flow $\mathcal{F}$, we wish to have a vector of end-to-end latency bound for the 12 methods evaluated in Section II-C, namely: $\mathbf{o}_\mathcal{F} = [D_\mathcal{F}^{m_1}, D_\mathcal{F}^{m_2}, \ldots, D_\mathcal{F}^{m_{12}}]$. Similar output vectors may be used for the servers' backlog bound.

IV. Numerical Evaluation

We evaluate in this section the accuracy of the proposed heuristic as introduced in Section III-C.

A. Evaluation as latency bound heuristic

We first assess in this section the precision of the predicted end-to-end latency bounds. Figure 4 illustrates the absolute relative difference between the predicted end-to-end delay bound and the bound given by the analytical method (named here *ground truth*). The overall median value is 2.5 %, with larger errors in case of predicting the output of PMOO.

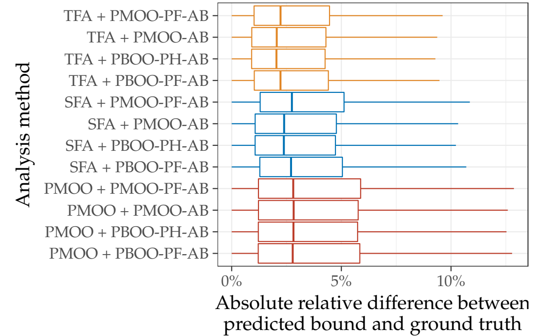


Figure 4: Precision of the predicted end-to-end latency bound

B. Evaluation as a proxy for analysis method selection

Based on the results from Section II-C, we introduce here an adaptive network analysis method, where a specific network analysis method is used given a flow of interest. This adaptive network analysis is illustrated in Algorithm 1.

Algorithm 1 Adaptive network analysis

```
for all flow of interest F in network N do
    netcalc_method ← select_netcalc_method(N, F)
    bound_F ← netcalc_method(N, F)
```

For the function *select_netcalc_method* in Algorithm 1, we define here the following strategies:

- *Global top 1*: we always use the network analysis method which provided the best median bound in Section II-C (ie. PMOO with SFA per-hop arrival bounding);
- *Global top 2*: we evaluate the two methods which provided the best median bounds in Section II-C (ie. PMOO with SFA per-hop and PMOO global arrival bounding) and select the method which produced the tightest bound;
- *PMOO and SFA*: both PMOO and SFA with SFA per-hop arrival bound are evaluated;
- *Per-topo best*: we use the method which provided the best median bound over the flows of the studied topology;
- *Fully random*: we randomly select among the 12 methods surveyed here, with the same probability for each method;
- *Weighted random*: same as in *Fully random* but with probability values set according a ranking of the methods;
- *ML top 1*: the heuristic from Section III-C is used for selecting the analysis according to the minimum predicted end-to-end delay bound;
- *ML top 2*: as in *ML top 1*, but two analyses are selected and the one producing the minimal tightest bound is kept.

We first evaluate in Figure 5 the ability of Algorithm 1 to produce the tightest per-flow end-to-end delay bound according to the different selection strategies previously listed. The machine learning based strategies outperform all the other strategies in Figure 5, with the ability to produce the tightest result for 88 % of the studied flows for the *ML top 2* strategy, outperforming all the other evaluated strategies.

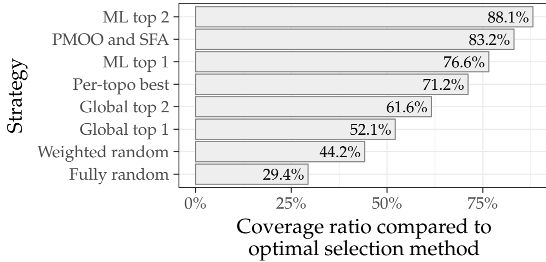


Figure 5: Ability of a selection method to produce the tightest end-to-end delay bound.

As in Figure 2(c), we evaluate in Figure 6 how much tightness is lost in case the produced end-to-end latency bound is not the tightest. Using the machine learning heuristic, we still get tight bounds compared to the other approaches listed earlier. This means that although the heuristic did not result in selecting the tightest network calculus analysis to use, the one which was selected produced a bound close to the tightest one.

Finally we evaluate in Figure 7 the gain in tightness of using an adaptive network analysis compared to only using a single analysis for all the evaluated topologies. In average, we see that we get a relative gain in tightness of around 12.79 % by using an adaptive network analysis based on machine learning, close to the optimal 13.03 %.

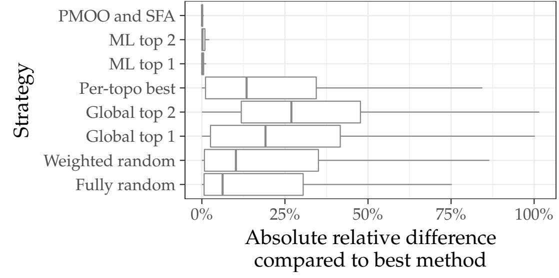


Figure 6: Relative difference between delay bound of a given method and the best method when the given method does not provide the best bound

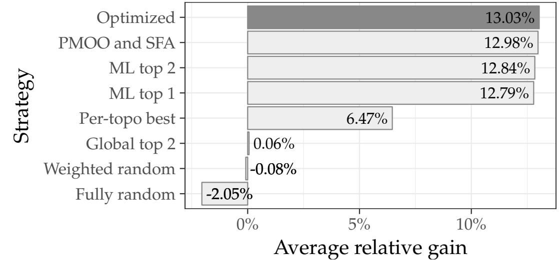


Figure 7: Relative gain in tightness compared to using only one method for all the evaluated topologies. "Optimized" corresponds here to the evaluation of the 12 network analyses.

C. Runtime

We evaluate here the runtime of using such heuristic in order to assess if the heuristic is of practical benefit regarding computation time. We evaluated the machine learning heuristic both on GPU (Nvidia GeForce GTX 1080 Ti) and CPU (Intel Xeon E3-1270). The network calculus analyses were run on the same CPU. The runtimes discussed here do not include the computation cost of training the neural network.

We compare in Figure 8 the average runtime per topology of the machine learning heuristic against the average runtimes of the different network analyses studied here. Compared to a single analysis, our heuristic is an order of magnitude faster on GPU. Compared to the sum of the runtimes of all the analyses, our heuristic is two orders of magnitude faster on GPU. This shows that our machine learning heuristic is both the best performing regarding accuracy as showed in Figures 5 and 7, but can also be used at little computational cost on GPU.

V. Related work

Identifying corner-cases of network calculus has already attracted some previous work. Schmitt et al. [9] showed that PMOO suffers from shortcomings given specific network configurations, and provided an optimization-based analysis that implements all three analysis principles at the same time. However, Kiefer et al. [14] showed that this optimization method suffers from vast computational effort. Bouillard et al. proposed in [4] another attempt to solve this challenge is using

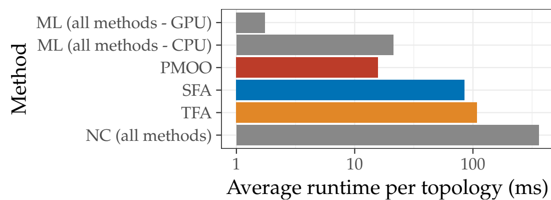


Figure 8: Average runtimes per topology of the different network analysis and arrival bounding methods, and the machine learning heuristic.

optimization-based analysis, but it was similarly showed by Bondorf et al. [15] to become computationally infeasible.

Various proposals have been made in order to make network calculus more computationally efficient. Luangsomboon et al. [16] proposed to use GPUs for computing fast convolution and deconvolution. While this approach provided efficient operations of the $(\min, +)$ algebra, the benefit in the case of network analysis is still to be determined. Bondorf et al. [15] recently proposed another approach based on exhaustive decomposition of network. Numerical evaluation showed that this method could reach bounds comparable to the ones from optimization-based methods at lower computational cost.

Neural networks for graphs has recently attracted a larger interest, and are generally based on the concept of message passing presented in Section III. They have been used in a variety of domains such as performance evaluation of networks with TCP flows [17], routing protocols [18], or basic logical reasoning tasks and program verification [12].

VI. Conclusion

Through a numerical evaluation on randomly generated networks of various network analysis methods from DNC, we showed and confirmed the existence of corner-cases, highlighting that the choice of which method to use given a network and a flow of interest is not trivial. This motivated our case for having a DNC heuristic able to predict which network analysis method will produce the tightest bound.

We contributed in this paper a novel heuristic for deterministic network calculus using graph-based deep learning. Our approach is based on the application of Graph Neural Networks and a mapping from feed-forward server graphs and the flows crossing them to graphs which can be used for training a neural network. We showed via a numerical evaluation that our approach is able to reach good accuracies and predict which network analysis will produce the tightest bounds. Finally, we evaluated the runtime of our heuristic and showed that it can be used at a small computational cost compared to traditional network analyzes.

Acknowledgments

The authors would like to thank Steffen Bondorf for his feedback on an early version of this paper. This work was supported by the German Federal Ministry of Education and Research (grant 16KIS0538, project DecADe), by the German-French Academy for the Industry of the Future, and the High-Performance Center for Secure Networked Systems.

References

[1] F. Geyer and G. Carle, "Network Engineering for Real-Time Networks: Comparison of Automotive and Aeronautic Industries Approaches," *IEEE Commun. Mag.*, vol. 54, no. 2, pp. 106–112, Feb. 2016.

[2] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001.

[3] J. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once," in *Proceedings of the 14th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems (MMB 2008)*, Mar. 2008, pp. 1–15.

[4] A. Bouillard, L. Jouhet, and E. Thierry, "Tight Performance Bounds in the Worst-Case Analysis of Feed-Forward Networks," in *INFOCOM 2010*. IEEE, Mar. 2010.

[5] C.-S. Chang, *Performance Guarantees in Communication Networks*. Springer, 2000.

[6] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Catching Corner Cases in Network Calculus – Flow Segregation Can Improve Accuracy," in *Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems*, Feb. 2018.

[7] S. Bondorf and J. B. Schmitt, "Calculating accurate end-to-end delay bounds – you better know your cross-traffic," in *Proceedings of the 9th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2015)*, Dec. 2015.

[8] ——, "The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus," in *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2014)*, Dec. 2014.

[9] J. B. Schmitt, F. A. Zdarsky, and M. Fidler, "Delay Bounds under Arbitrary Multiplexing: When Network Calculus Leaves You in the Lurch...," in *Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*, ser. INFOCOM 2008, Apr. 2008, pp. 1669–1677.

[10] M. Gori, G. Monfardini, and F. Scarselli, "A New Model for Learning in Graph Domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, ser. IJCNN'05, vol. 2. IEEE, Aug. 2005, pp. 729–734.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[12] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks," in *Proceedings of the 4th International Conference on Learning Representations*, ser. ICLR'2016, Apr. 2016.

[13] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," Jun. 2014.

[14] A. Kiefer, N. Gollan, and J. Schmitt, "Searching for tight performance bounds in feed-forward networks," *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, pp. 227–241, 2010.

[15] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 1, no. 1, p. 34, 2017.

[16] N. Luangsomboon, R. Hesse, and J. Liebeherr, "Fast Min-plus Convolution and Deconvolution on GPUs," in *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS 2017, Dec. 2017.

[17] F. Geyer, "Performance Evaluation of Network Topologies using Graph-Based Deep Learning," in *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS 2017, Dec. 2017, pp. 20–27.

[18] F. Geyer and G. Carle, "Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning," in *Proceedings of the 2018 SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, ser. Big-DAMA 2018. Budapest, Hungary: ACM, Aug. 2018, pp. 40–45.

A.1.5 DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks

This work was published in *Proceedings of the 38th IEEE International Conference on Computer Communications*, 2019 [62].

DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks

Fabien Geyer*
Technical University of Munich | Airbus CRT
Munich, Germany

Steffen Bondorf†
NTNU – Norwegian University of Science and Technology
Trondheim, Norway

***Abstract*—Network calculus computes end-to-end delay bounds for individual data flows in networks of aggregate schedulers. It searches for the best model bounding resource contention between these flows at each scheduler. Analyzing networks, this leads to complex dependency structures and finding the tightest delay bounds becomes a resource intensive task. The exhaustive search for the best combination of contention models is known as Tandem Matching Analysis (TMA). The challenge TMA overcomes is that a contention model in one location of the network can have huge impact on one in another location. These locations can, however, be many analysis steps apart from each other. TMA can derive delay bounds with high degree of tightness but needs several hours of computations to do so. We avoid the effort of exhaustive search altogether by predicting the best contention models for each location in the network. For effective predictions, our main contribution in this paper is a novel framework combining graph-based deep learning and Network Calculus (NC) models. The framework learns from NC, predicts best NC models and feeds them back to NC. Deriving a first heuristic from this framework, called DeepTMA, we achieve provably valid bounds that are very competitive with TMA. We observe a maximum relative error below 6 %, while execution times remain nearly constant and outperform TMA in moderately sized networks by several orders of magnitude.**

I. Introduction

A. Motivation

Deterministic performance bounds have seen many applications in modern systems and a wide range of network calculus-based solutions have been proposed. Network Calculus (NC) can be applied to ensure deadlines in networks for x-by-wire applications [1] as well as SDN-enabled networks [2], for safety-critical production systems [3], or both of these [4]. Moreover, NC solutions have been proposed for highly dynamic environments. E.g., admission control in self-modeling sensor networks [5] or systems providing customers with service level agreements [6] for, among others, storage access [7]. Other recent examples where dynamic events may often cause changes are cache networks [8] and cloud computing [9]. These areas benefit from fast computations of tight performance bounds. The literature provides one-shot analyses for topology-agnostic bounds [10] or bounds that hold

* This work was supported by the German-French Academy for the Industry of the Future.
†This work was partially carried out during the tenure of a Carl-Zeiss Foundation fellowship in the DISCO Labs at TU Kaiserslautern, Germany, and partially during the tenure of an ERCIM ‘Alain Bensoussan’ Fellowship Programme at NTNU Trondheim, Norway.

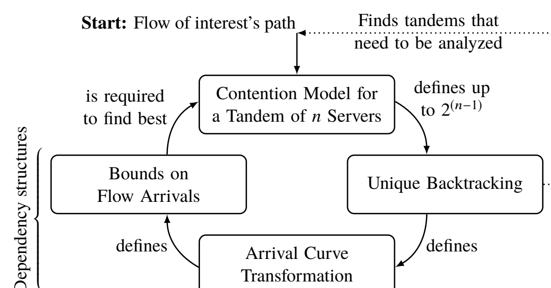


Figure 1: Dependency cycle defining current NC analyses.

for the specification's worst case [4]. Yet, these attempts are ultimately paid for with wasted resources. Our approach does not compromise on bound quality by providing a fast analysis that considers all details of the analyzed network.

B. Background

In network calculus, a network needs to be modeled by servers (e.g. queues or packet schedulers) whose forwarding capabilities are lower bounded by service curves. These curves are derived for each server's respective aggregate scheduler. Data flows traverse sequences of servers where they compete for resources with other flows. Multiplexing and reordering in queues can occur arbitrarily but deterministic bounds can be computed as data arrivals are bounded by arrival curves. The arrival curves are, however, only known at a flow's first server.

Given such a network model, the NC analysis computes a bound on an individual flow's end-to-end delay. This flow is known as flow of interest (foi) and NC must derive a model for resource contention from this flow's point of view. NC offers multiple network analysis methods to derive contention models that discriminate against the foi. These alternatives are all proven to result in valid delay bounds for the foi. But there is not a single-best contention model that can be expressed with NC, not even on a simple tandem of servers. All the worse for NC, it needs to bound the impact of all transformations of all flows' arrival curves up to the location of contention to rank the contention models. Curve transformations, in turn, require to backtrack all flows, either in an aggregate or separated by worst-case priority assumptions. Different contention models require different flow aggregation/separation assumptions and

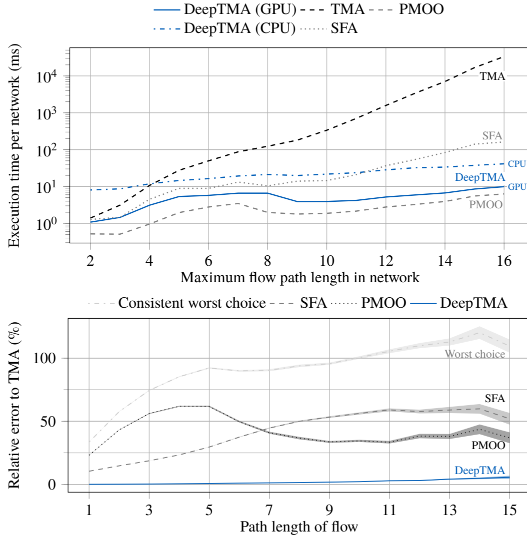


Figure 2: Comparison of DeepTMA to existing NC heuristics.

the resulting structures expressing dependencies of NC operations become unique. This cycle is shown in Figure 1. An analysis must execute at least one complete recursion, terminating upon reaching all backtracked flows' sources.

It was shown that it is possible to exhaustively derive all dependency structures and rank each contention model on each tandem occurring in a network analysis. This is known as the Tandem Matching Analysis (TMA) [11]. It achieves high degrees of delay bound tightness by enumerating all contention models upstream from the foi. Thus, the best model for a downstream location and flow can be found. In other words, TMA unwinds all loops that can be taken in the cycle in Figure 1. This is very costly. The amount of alternative contention models on a single tandem of n servers is $2^{(n-1)}$. TMA provides a recursive algorithm whose execution time can exceed several hours, e.g., when analyzing networks with >1000 servers and four times as many flows on many-core platforms that compromise on per-core performance [11].

In this paper, we present the deep-learning assisted TMA, DeepTMA, that predicts the best contention model with high efficacy, resulting in a high degree of delay bound tightness although we only start a single backtracking in Figure 1. Single backtrackings have been attempted before, yet, we are the first to achieve considerably faster execution times than TMA without considerably compromising on delay bound tightness.

C. Contributions

While we focus our evaluations on the novel DeepTMA heuristic for NC's tandem matching analysis, we contribute an entire underlying framework that combines the theories of NC [12] and a graph-based deep learning, namely Graph Neural Networks (GNNs) [13], as well as two of their respective tools [14, 15]. DeepTMA achieves the following properties:

a) Deterministic bounds: We learn from NC and feed predictions back to NC. We predict the best choices for decisions made during the NC analyses. NC stays in control and guarantees provably correct bounds.

Our deep learning framework does not learn to predict a delay bound but it predicts the most important decisions within an analysis, the contention models. Compared to directly predicting a flow's delay bound, our approach always guarantees for a valid worst-case bound as we continue to apply the proven NC operations in their valid orders.

b) Fast execution times and high tightness: Figure 2 shows first benchmarking results of DeepTMA. We compare against TMA and the established SFA [12] and PMOO [16] heuristics of NC. These are fast as they greedily decide on a single contention model, ignoring arrival and service curves. DeepTMA from our framework is minimally slower than PMOO but faster than SFA and TMA. Moreover, recent work [17] showed that the TMA cannot be parallelized easily and a speedup of only one order of magnitude was observed. In terms of delay bound tightness (relative error to TMA), all heuristics outperform a consistent worst choice of contention models. DeepTMA-derived delay bounds are tightest among these heuristics, deviating from TMA by no more than 6 % in our experiments.

DeepTMA efficacy beating SFA and PMOO in the cost/tightness-tradeoff is necessary, yet, by no means sufficient to conclude that our deep-learning assisted analysis framework is the best alternative to create heuristics. SFA and PMOO were created a decade before TMA, i.e., they never benefited from advances that resulted in TMA. Therefore, we base our statement on numerical evaluations benchmarking against newly contributed non-deep-learning TMA heuristics from the NC framework.

c) Train once, apply infinitely often: Naturally, we only train our machine learning part once before using its predictions in DeepTMA. While we chose a reasonably large range of parameters for arrival and service curves to learn from, we restricted our dataset to simple topologies (tandems and sink trees). The results shown above are achieved by predicting the best contention model for bounding each flow's delay in different, independently created tandems and sink trees.

d) Portability: While we combined two existing tools whose dependencies must be met, our combination of both theories enforces no dependencies. It is generally portable to any platform used in an area mentioned in the beginning. For instance, Figure 2 shows results for execution on CPU or GPU. Moreover, efficient deep learning libraries are becoming increasingly available in a variety of programming languages.

D. Outline

The remainder of the paper is organized as follows: Section II presents the NC theory, covering modeling and the TMA, that we will express as a graph analysis task and combine with GNNs in Section III. In Section IV we present the combination of tools and the generation of a dataset to learn from. Section V provides new NC heuristics in order

to benchmark DeepTMA against modern non-deep-learning approaches in Section VI. Section VII presents the related work on our research direction for network calculus and graph neural networks before Section VIII concludes our work and gives an outlook.

II. Network Calculus

NC models resource provision and demand with non-negative, wide-sense increasing functions from the set

$$\mathcal{F}_0 = \left\{ f : \mathbb{R} \to \mathbb{R}^+_\infty \mid f(0) = 0,\ \forall s \leq t : f(s) \leq f(t) \right\},$$

$\mathbb{R}^+_\infty := [0, +\infty) \cup \{+\infty\}$. Functions of $\mathcal{F}_0$ are also used for the bounding curves of NC. Arrival curves upper bound data arrivals and service curves lower bound forwarding guarantees.

Definition 1 (Arrival Curve): Let the data arrivals of a flow over time be characterized by function $A(t) \in \mathcal{F}_0$, where $t \in \mathbb{R}^+_\infty$. Then, an arrival curve $\alpha(d) \in \mathcal{F}_0$ for $A(t)$ must fulfill

$$\forall t \forall d,\ 0 \leq d \leq t : A(t) - A(t-d) \leq \alpha(d),$$

i.e., it must bound the flow's data arrivals in any duration d.

Definition 2 ((Strict) Service Curve): If, during any period with backlogged data of duration d, a server s with input function A guarantees an output of at least $\beta(d) \in \mathcal{F}_0$, then it is said to offer a (strict) service curve β.

The network calculus was cast in a (min,+)-algebra [12] with the following operations:

Definition 3 ((min,+) Operations): Network calculus applies $(\min, +)$-algebraic operations to compute curve transformations bounding the worst-case outcome of certain scenarios:

- *Flow Aggregation:* $(\alpha_1 + \alpha_2)(d) = \alpha_1(d) + \alpha_2(d)$
- *Server Crossing:* $(\alpha \oslash \beta)(d) = \sup_{u \geq 0} \{\alpha(d+u) - \beta(u)\}$
- *Residual Service:* $(\beta \ominus \alpha)(d) = \sup_{0 \leq u \leq d} \{\beta(u) - \alpha(u)\}$
- *Server Concat.:* $(\beta_1 \otimes \beta_2)(d) = \inf_{0 \leq u \leq d} \{\beta_1(d-u) + \beta_2(u)\}$

where $\alpha, \alpha_1, \alpha_2$ are arrival and β, β_1, β_2 are service curves.

These curve transformations guarantee for deterministic results. For a network such as the tandem shown in Figure 3, there are multiple valid orders of operations. Each corresponds to a model of contention that imposes a dependency structure. That structure, in turn, defines contention models upstream.

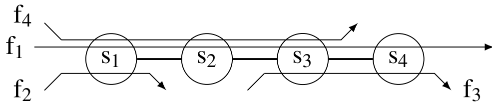


Figure 3: Example tandem network in the NC model.

Definition 4 (Contention Model): The network calculus contention model for a tandem of servers defines its orders of operations that provide a residual service guaranteed to a flow crossing said tandem. Any concatenation of sub-tandem residual service is a valid contention model for the tandem.

Suppose f_3 in Figure 3 is the flow of interest to be analyzed. Its *tandem decompositions* are defined by subtandem-separating *cuts* located between crossed servers:

all cuts: decomposition into 1-server tandems s_3 and s_4 or
no cuts: entire 2-server tandem consisting of s_3 and s_4.

On any tandem, any number and placement of cuts results in a valid tandem decomposition. Their exhaustive enumeration is known as Tandem Matching Analysis (TMA) [11]. The two cases shown here are special. The former corresponds to the classical Separated Flow Analysis (SFA) [12]. It concatenates per-server residual service bounds by computing $(\beta_3 \ominus (\alpha_{3,1} + \alpha_{3,4})) \otimes (\beta_4 \ominus \alpha_{4,1})$, where the first index denotes server location and the second one of arrival curves is the flow id. The latter contention model is known as the Pay Multiplexing Only Once (PMOO) [16] analysis that computes f_3's residual service for the concatenated tandem: $(\beta_3 \otimes \beta_4) \ominus' \{\alpha_{3,1}, \alpha_{3,4}\}$. Note the adapted residual service operation from [11] and the set of separated flows it subtracts.

Next, we need to bound the arrivals of flows to an analyzed tandem as required for the residual service curve computation under a specific contention model.

Definition 5 (Dependency Structure): A dependency structure is a set of sequences that bound arrival curve transformations up to a tandem of servers.

The dependency structure is subject to the contention model's requirements regarding flow aggregation, separation and duplication. It is created by unwinding the cycle shown in Figure 1; potentially under contention modeling restrictions (SFA or PMOO). In our example, SFA can bound f_1 and f_4 aggregately up to s_3, i.e., on the tandem of servers s_1 and s_2. Note, that either of the two decompositions as above can be best due to the interference of f_2. Additionally, SFA needs a separate arrival curve for f_1 at s_4 – it is not possible to separate flows after their aggregate crossing of a server was bounded and we do not assume additional network elements alleviating this problem like per-flow shapers [12] or interleaved traffic regulators [18]. PMOO depends on separate bounds for f_1 and f_4 for the same reason. Both flows cross the s_1,s_2-tandem and can thus benefit from either contention model due to f_2. Yet, PMOO does not check the SFA contention model / tandem decomposition. Last, note that TMA computes results for all these contention models and dependency structures to find the best one. Depending on the employed hardware, executing the TMA can take multiple hours. For example, analyzing a network with about 1500 servers and four times as many flows was shown to take close to 2 hours on a compute server [11]. Therefore, we aim to avoid this huge effort with predictions.

III. Graph Neural Network for NC

We develop our DeepTMA heuristic in this section. It is based on the concept of Graph Neural Network (GNN) introduced in [13, 19]. The goal of DeepTMA is to predict the best tandem decompositions, i.e., contention models, to use in TMA. We define networks to be in the NC modeling domain and to consist of servers, crossed by flows. We refer to the model used in GNN as graphs. The main intuition is to transform the networks into graphs. Those graph representations are then used as inputs for a neural network architecture

able to process general graphs, which will then predict the tandem decomposition resulting in the best residual service curve. Our approach is illustrated in Figure 4. Since the delay bounds are still computed using the formal network calculus analysis, they inherit their provable correctness.

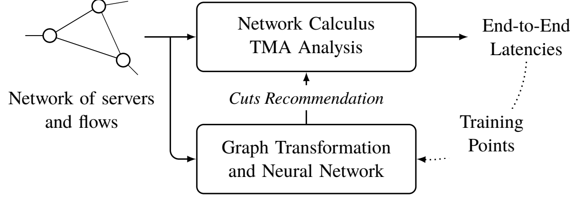


Figure 4: Overview of the proposed approach.

A. Overview of Graph Neural Networks

In this section, we detail the neural network architecture used for training neural networks on graphs, namely the family of architectures based on GNNs [13, 19].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $(v, u) \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and output values for node v. The concept behind GNNs is called *message passing*, where hidden representations of nodes $\mathbf{h}_v$ are iteratively passed between neighboring nodes. Those hidden representations are propagated throughout the graph using multiple iterations until a fixed point is found. The final hidden representation is then used for predicting properties about nodes. This concept can be formalized as:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_u^{(t-1)} \;\middle|\; u \in \text{NBR}(v)\right\}\right) \tag{1}$$

$$\mathbf{o}_v = out\left(\mathbf{h}_v^{(t\to\infty)}\right) \tag{2}$$

$$\mathbf{h}_v^{(t=0)} = init\left(\mathbf{i}_v\right) \tag{3}$$

with $\mathbf{h}_v^{(t)}$ representing the hidden representation of node v at time t, $aggr$ a function which aggregates the set of hidden representations of the neighboring nodes $\text{NBR}(v)$ of v, out a function transforming the final hidden representation to the target values, and $init$ a function for initializing the hidden representations based on the input features.

The concrete implementations of the $aggr$ and out functions are feed-forward neural networks (FFNN), with the addition that $aggr$ is the sum of per-edge terms [19], such that:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_{\text{NBR}(v)}^{(t-1)}\right\}\right) = \sum_{u\in\text{NBR}(v)} f\left(\mathbf{h}_u^{(t-1)}\right) \tag{4}$$

with f a FFNN. For $init$, a one-layer FFNN is used to fit the input features to the dimensions of the hidden representations.

Gated Graph Neural Networks (GGNN) [20] were recently proposed as an extension of GNNs to improve their training. This extension implements f using a memory unit called Gated Recurrent Unit (GRU) [21] and unrolls Equation (1) for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing. The neural network architecture is illustrated in Figure 5.

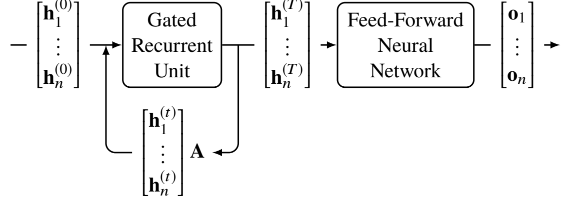


Figure 5: Gated-graph neural network architecture.

Formally, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \tag{5}$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z\mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \tag{6}$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r\mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \tag{7}$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W}x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \tag{8}$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \tag{9}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}$ and $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}$ are trainable weight matrices, and $\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining the edges in the graph $\mathcal{G}$.

Equation (5) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for GNNs in Equations (1) and (4). Equations (6) to (9) correspond to the mathematical formulation of a GRU cell [21], with Equation (6) representing the GRU reset gate vector, Equation (7) the GRU update gate vector, and Equation (9) the GRU output.

In order to propagate the hidden representations throughout the complete graph, a fixed number of iterations of Equations (6) to (9) are performed. This extension has been shown to outperform standard GNN which require to run the iteration until a fixed point is found.

We also extended our neural network architecture with an attention mechanism similar to the one proposed in [22]. Thus, the neural network can give preference to some neighbors over other ones via a trained function. For each edge (v, u) in the graph, we define a weight parameter $\rho_{v,u}^{(t)}$ depending on the concatenation of $\mathbf{h}_v^{(t)}$ and $\mathbf{h}_u^{(t)}$:

$$\rho_{v,u}^{(t)} = \sigma\left(\mathbf{W}_a\left\{\mathbf{h}_v^{(t)}, \mathbf{h}_u^{(t)}\right\} + \mathbf{b}_a\right) \tag{10}$$

with trainable weights $\mathbf{W}_a$ and bias parameters $\mathbf{b}_a$. Equation (4) can then be rewritten as

$$\mathbf{h}_v^{(t)} = \sum_{u\in\text{NBR}(v)} \rho_{v,u}^{(t-1)} f\left(\mathbf{h}_u^{(t-1)}\right). \tag{11}$$

B. Application to TMA

In order to apply the concepts described in Section III-A to a network calculus analysis, we model NC's network into a graph. Figure 6 illustrates this graph encoding on the network from Figure 3.

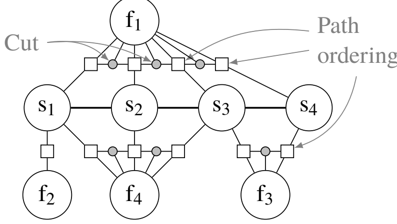


Figure 6: Transformed network of Figure 3 to the graph model.

Each server is represented as a node in the graph, with edges corresponding to the network's links. Each flow is represented as a node in the graph, too. In order to encode the path taken by a flow in this graph, we use edges to connect the flow to the servers it traverses. Since those edges do not encode the order in which those servers are traversed, so-called *path ordering* nodes are added to edges between the flow node and the traversed server nodes. This property is especially important in the TMA since the order, and hence position of cuts, has a large impact on dependency structures. In order to represent these TMA cuts, each potential cut between pairs of servers on the path traversed by the flow is represented as a node. This cut node is connected via edges to the flow and to the pair of servers it is associated to.

In addition to a categorical encoding of the node type (i.e., server, flow, path ordering or cut), the input features of each node in the graph is as follows:

- For each server s, parameters of its service curve $\beta_s(d) = \max\{0, rate_s \cdot d - latency_s\}$ are used: $[rate_s, latency_s]$
- For each flow f, parameters of its arrival curve $\alpha_f(d) = \{rate_f \cdot d + burst_f\}_{\{d>0\}}$ (i.e., $\alpha_f(d) = 0$ for $d \leq 0$) are used: $[rate_f, burst_f]$
- For each path ordering p, the hop count is encoded as a categorical one-hot vector: $[hop = 1, \ldots, hop = n]$
- Finally, cut nodes do not have input features

Note that in case more complex arrival or service curve shapes than affine curves [23] are studied, those input features can be extended to represent the additional curve parameters. Last note that edges have no features in this graph encoding.

Since the goal of our machine learning approach is to predict which tandem decomposition will result in the tightest bound, only the nodes presenting cuts have output features. We encode this problem as a classification problem, namely each cut node has to be classified in two classes: perform a cut between the pair of servers it is connected to or not: $[cut, \overline{cut}]$. The overall prediction to be fed back, i.e., the selection of one out of TMA's potential decompositions for a given foi's path, is defined by the set of all cut classifications for this path.

IV. Implementation and Dataset Generation

A. Technical Implementation

We implemented DeepTMA using Tensorflow. For the purpose of computational efficiency, passing of hidden representation between neighboring nodes was implemented with sparse operations using the graph's adjacency list instead of the graph's adjacency matrix requiring dense operations. The recursion from Equation (1) was unrolled for a fixed number of iteration according to the diameters of the analyzed graphs. Table I illustrates the size of the different layers used here.

Layer	NN Type	Size
$init$	FFNN	$(21, 160)_w + (160)_b$
Memory unit	GRU cell	$(320, 320)_w + (320, 160)_w + (480)_b$
Edge attention	FFNN	$(320, 1)_w + (2)_b$
out hidden layers	FFNN	$2 \times \{(160, 160)_w + (160)_b\}$
out final	FFNN	$(160, 2)_w + (2)_b$
	Total:	209 766 parameters

Table I: Size of the different layers used in the GGNN. Indexes represent respectively the weights (w) and biases (b) matrices.

We analyzed each network with the DiscoDNC [14] version 2.4. A tandem decomposition is always executed for a flow of interest. But instead of the residual service curves, we use the delay bounds for the foi as caused by all decompositions in order to rank them. This is because the former potentially faces problems in the case of lost service curve strictness.

B. Dataset Generation

In order to train our neural network architecture, we randomly generated a set of topologies, tandems like in Figure 3 and tree topologies. For each created server, a rate latency service curve was generated with uniformly random rate and latency parameters. A random number of flows with random source and sink servers was added. Note that in our topologies, there cannot be cyclic dependency between the flows. For each flow, a token bucket arrival curve was generated with uniformly random burst and rate parameters. All curve parameters were normalized to the $(0, 1]$ interval. In total, 100 000 different networks were generated, with a total of more than 2 million flows, and close to 60 million tandem decompositions. Half of the networks were generated following tandem topologies, and half following tree topologies. Table II summarizes different statistics about the generated dataset. The dataset is available online[1] to reproduce our learning results.

Parameter	Min	Max	Mean	Median
# of servers	2	41	14.2	12.0
# of flows	1	63	23.0	18.0
# of flows per server	1	44	5.8	4.6
# of tandem combinations	2	113 100	596.2	134.0
# of tandem combination per flow	2	32 768	25.9	4.0
# of nodes in analyzed graph	6	717	159.0	127.0

Table II: Statistics about the generated dataset.

[1] https://github.com/fabgeyer/dataset-infocom2019

V. TMA Heuristics from the NC Framework

As we will detail in Section VII, there is no other combination of NC and deep learning for deterministic performance analysis. Nor is there any other combination of NC or deep learning with a third methodology for fast delay bounding. To benchmark DeepTMA, we present three new heuristics for the choice of TMA's tandem decompositions. All are derived from the NC framework to showcase its potential to find the tightest end-to-end delay bound without exhaustive analysis.

A. RND: Random Choice of Tandem Decomposition

The simplest heuristic is to select multiple alternative tandem decompositions randomly following a uniform distribution. Given any n-server tandem, starting with the foi's path as shown in Figure 1, RND only selects $n' \ll 2^{(n-1)}$ decompositions. I.e., the RND heuristic randomly samples a small part of TMA's search space per tandem in the analysis. The remainder of the analysis follows the standard NC proceeding.

B. PLH: Path Length of Flows up to Location of Interference

Due to the exponential growth of the number of tandem decompositions, the chance of randomly selecting the tandem decomposition resulting in the tightest bound decreases exponentially with the number of servers traversed by the foi. In this second heuristic, we use the intuition that the probability of a cut depends on the cross-flow at each server and the fact that the arrival curve of cross-flows depends on the path traversed. In order to define it, we use h, the number of servers that each cross-flow crossed before reaching this location, and empirically fit the probability $\Pr(cut|h_{\text{avg}})$ with h_{avg} the average of h over all cross-flows. In homogeneous sink-tree networks, this heuristic can even obtain a precise ranking of flow arrival curves as well as the relative differences between them due to the lack of flow demultiplexing [5].

C. HCH: Hop Count Heuristic

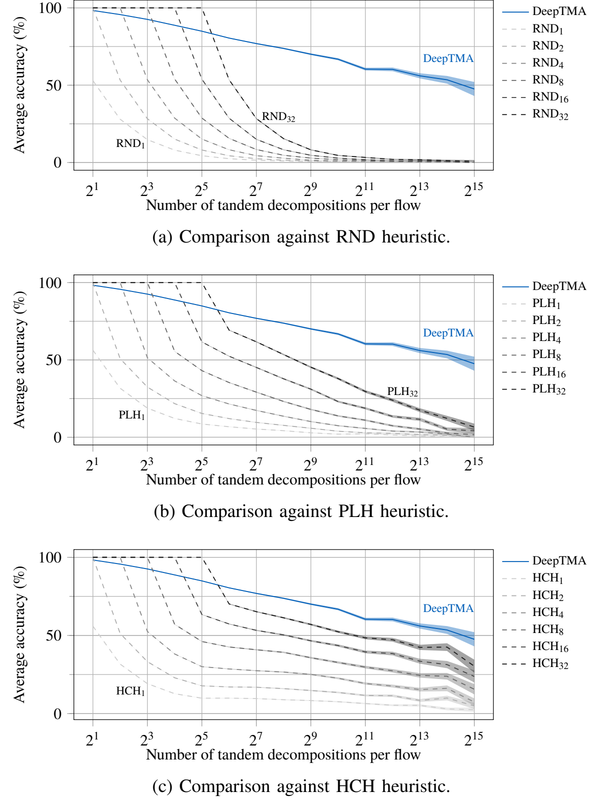
This last heuristic is based on the probability of cutting a tandem according the number of traversed servers as observed in our data set. In order to correctly parametrize this location, we empirically fit a distribution $\Pr(cut|l,p)$ predicting the best cut for each path length l and cut position p. The procedure for generating a tandem cut is illustrated in Algorithm 1. As in Section V-A, multiple tandem decompositions may be generated and evaluated.

Algorithm 1 Decomposition using experimental probability.

$v \leftarrow [c_1, \ldots, c_l] \sim \mathcal{U}(0,1)^l$
cuts $\leftarrow \mathbb{I}\,(v \leq [\Pr(cut|l,1), \ldots, \Pr(cut|l,l)])$
($\mathbb{I}$ is the indicator function)

VI. Numerical Evaluation

We evaluate in this section DeepTMA against the heuristics presented above. Via a numerical evaluation, we illustrate the tightness and execution time of DeepTMA and highlight its usability for practical use-cases.



(a) Comparison against RND heuristic.

(b) Comparison against PLH heuristic.

(c) Comparison against HCH heuristic.

Figure 7: Accuracy of DeepTMA and the new heuristics.

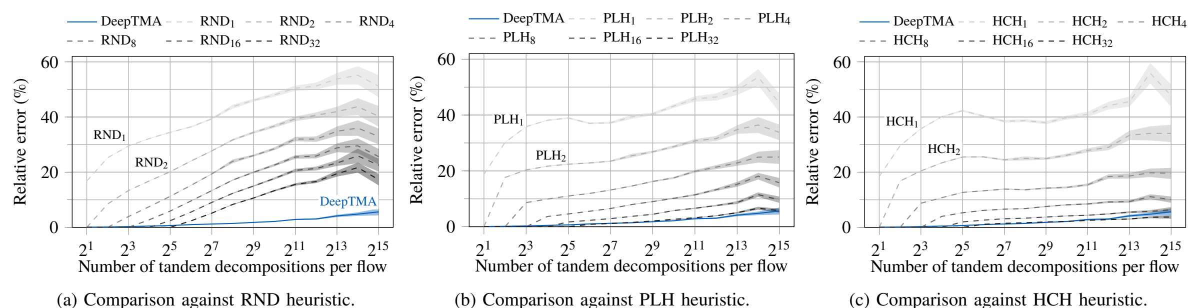
A. Prediction Accuracy

Since multiple tandem decompositions may be valid and since we know the tightest bounds from TMA, we define the accuracy for a given foi and a given method as 1 if the tandem decomposition predicted by the method resulted in the tightest delay bound, and 0 otherwise. We evaluate in Figure 7 the outcome of the different heuristics evaluated on our dataset.

Figure 7a compares DeepTMA against the RND heuristic presented in Section V-A, namely random choices of tandem decomposition. We also evaluate the case where multiple random tries are evaluated and the one leading to the tightest delay bound is kept (labeled by the index in the figure). We note in Figure 7a that DeepTMA achieves average accuracies larger than 50 % even for flows where the possible number of tandem decomposition goes up to 32 768. Compared to this heuristic, DeepTMA achieves much better accuracies for flows with a larger number of hop.

Figure 7b compares DeepTMA against the PLH heuristic presented in Section V-B, namely the cross-flow statistics heuristic. This heuristic achieves better accuracy compared to the previous one, but it still fails to reach good accuracy for networks with a large number of cuts.

Finally, Figure 7c compares DeepTMA against the HCH



(a) Comparison against RND heuristic. (b) Comparison against PLH heuristic. (c) Comparison against HCH heuristic.

Figure 8: Relative error of DeepTMA and the heuristics presented in Section V.

heuristic presented in Section V-C, namely the hop count statistics heuristic. We notice that this heuristic achieves better accuracy for networks with a larger number of hops compared to the two previous heuristics. Nevertheless, DeepTMA still achieves better results for larger topologies.

Overall, that DeepTMA achieves better prediction accuracy than the pure NC heuristics. This indicates that the choice of tandem decomposition is more complex than captured by indicators such as hop count or cross-flow statistics.

B. Relative Error

While we highlighted in the previous section that the accuracy of our machine learning approach diminishes as the number of hops becomes larger, we investigate in this section the resulting loss of tightness in case a non-optimal tandem decomposition was selected. In order to quantitatively evaluate this loss of tightness, we use the relative error, defined as

$$relative\ error_{\text{foi}} = \frac{delay_{\text{foi}}^{\text{heuristic}} - delay_{\text{foi}}^{\text{TMA}}}{delay_{\text{foi}}^{\text{TMA}}} \tag{12}$$

Since TMA always produces the tightest delay bound among the evaluated heuristics, this relative error is always positive.

Figure 8 compares DeepTMA against the other heuristics. Although the accuracy of DeepTMA dropped to 50 % for larger networks, the impact of these failures to predict the optimal decomposition only results in a relative error below 6 %. The results and comparison between DeepTMA and the other heuristics are in line with those presented in Figure 7. Only PLH_{32} and HCH_{32} are able to achieve a relative error similarly small as DeepTMA, yet, at a much larger computational cost since 32 different tandem combinations and their entire dependency structures have to be evaluated every time.

C. Execution Times

In order to understand the practical applicability of our heuristic, we evaluate in this section its execution time in different settings. We define and measure the execution time per network as the total time taken to process N networks and all its flows divided by N, without including the startup time or the time taken for initializing the network data structures.

Since DeepTMA can be executed on either CPU or GPU, we first compare both platforms and their affinity at parallelization in Figure 9. A Nvidia GTX 1080 Ti was used for the measurements on GPU, and an Intel Xeon E3-1270 v6 (at 3.80 GHz) for the ones on CPU. We first notice that the execution time grows close to linearly with the size of the network, both on CPU and GPU, which is explained by the iterations of message passing illustrated in Equation (5) according to the diameter of the studied graph. Execution on GPU results in faster computation compared to CPU for networks larger than two hops, mainly due to the better ability of GPUs of parallelizing the numerical operations used in neural networks.

Since both platforms offer multiple cores for parallel execution of multiple processes, we investigate the effect of batching, namely analyzing multiple networks in parallel. Parallelization of the mathematical operations described in Section III is automatically performed by Tensorflow. We present in Figure 9 the execution time without any batching – namely only one network is processed at once – and with batching, where the heuristic processes 64 networks at once. On both platforms, batching results in a reduction of processing time, which is relevant in use-cases where multiple network configuration have to be processed.

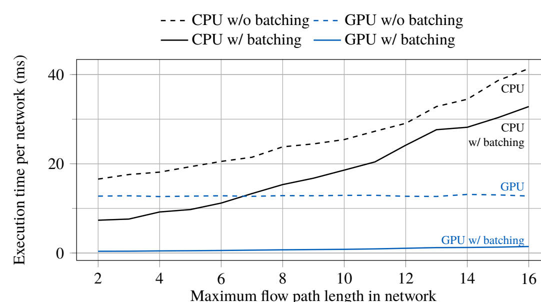


Figure 9: Execution time of the cut recommendation part of DeepTMA, executed on CPU or GPU, without batching or batch sizes of 64 networks.

In addition, we measured the execution time of TMA using

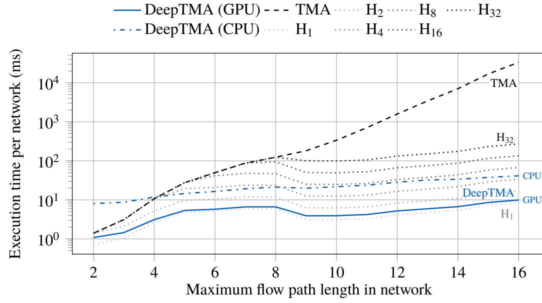


Figure 10: Execution times per topology for TMA, DeepTMA and heuristics H using n tandem decompositions (H_n).

DiscoDNC. The same CPU was used for running DiscoDNC, with Oracle's HotSpot JVM version 1.8.

Whereas Figure 9 provides insight on the computational cost of DeepTMA, Figure 10 compares it to a generalized version of the heuristics presented in Section V. Since the selection of tandem decompositions is a fast operation in all three pure NC heuristics, in particular compared to the other required operations, we only illustrate the execution time of a generic heuristic H_n selecting n decompositions per tandem. As all analyses ultimately use the DiscoDNC for the derivation of bounds, comparing the average execution times of H_n and DeepTMA (with batching), we can also judge the increase of computational effort due to our deep learning-based predictions. As expected, TMA execution times grow exponentially and H_n heuristics' execution times coincide with TMA as long as their n-value causes an exhaustive search, too. An entirely CPU-bound DeepTMA analysis is slowest in very small networks where the exhaustive enumeration of TMA is easily possible to execute. Starting at a maximum flow path length of 4, it mostly performs between H_4 and H_8. Yet, we saw in Sections VI-A and VI-B that RND_i, PLH_i and HCH_i, $i \in 4, 8$ are outperformed by DeepTMA. DeepTMA leveraging GPU technology for predictions only adds very small execution times to H_1 while achieving vastly better bounds. Compared to TMA, we can observe a measured differences in execution time growing up to four orders of magnitude.

Last, note the comparison between DeepTMA, TMA, SFA and PMOO that was already presented in Figure 2 as it highlights efficiency compared to established NC analyses.

VII. Related Work

A recent survey [24] about existing applications of machine learning to formal verification shows that this combination can accelerate formal methods, e.g., theorem proving, model-checking or SAT-SMT problems. As we show, NC has been combined with other methods, too. So have GNNs with formal verification. Yet, we are the first to combine both TMA and GNN into a framework for deterministic performance analysis.

1) Network Calculus Combined with Other Methodologies: The (min,+)-algebraic NC provides deterministic modeling and analysis techniques. It has seen various efforts to extend NC's capabilities. For instance, the underlying (min,+) algebra can be exchanged for (min,×) for fading channel analysis [25] or for (max,+) to better fit discrete event systems [26]. Moreover, a common model for NC and event stream theory has been developed [27] and state-based system modeling can be integrated by pairing NC with timed automata [28].

NC has been used to describe component models commonly found in real-time systems [29]. Delay bounds can then be derived from a combination of component characteristics and the network calculus model. For example, knowledge about the busy period of a greedy processing component has been used to speed up NC computations [30].

An optimization formulation has been derived from the NC model that computes tight bounds in networks without assumptions on the multiplexing of flows [31]. It first derives the dependencies between busy periods of servers in order to partially order the mutual impact of flows. The tight analysis requires to expand this order to all compatible total orders. This is, however, computationally infeasible. A heuristic was proposed. It uses the initially derived partial order but it, too, was shown to become computationally infeasible [11].

Recent work uses machine learning to estimate service curves from measurements [32]. In contrast to our work, this interfacing via service curves cannot compute provably correct bounds on the worst-case flow delays due to uncontrollable uncertainties introduced by measurements and machine learning.

2) Deep Learning for Graphs and Formal Verification: GNNs were first introduced in [13, 19], a concept subsequently refined in recent works. GGNNs [20] extended this architecture with modern practices by using GRU memory units [21]. Message-passing neural network were introduced in [33], with the goal of unifying various GNN and graph convolutional concepts. [22] formalized graph attention networks, which enables to learn edge weights of a node neighborhood.

These concepts were applied to many domains where problems can be modeled as graphs: chemistry with molecule analysis [34, 33], jet physics and elementary particles [35], prediction of satisfiability of SAT problems [36], or basic logical reasoning tasks and program verification [20]. For computer networks, they have recently been applied to prediction of delay bounds [37] and performance evaluation of networks with TCP flows for predicting average flow bandwidth [15, 38].

VIII. Conclusion

We contribute a new framework that combines network calculus and deep learning. The first heuristic created with our framework is the DeepTMA, deep learning-assisted TMA, a fast network analysis for deterministic end-to-end delay bounds. It solves the main bottleneck of the existing TMA, namely its exponential execution time growth with network size, by using predictions for effectively selecting the contention models in the network calculus analysis. Our work is based on a transformation of the network of servers and flows crossing them into a graph which is analyzed using

Graph Neural Networks. Via a numerical evaluation, we show that our heuristic is accurate and produces end-to-end bounds which are almost as tight as TMA. DeepTMA is as fast as or faster than previously widespread methods – namely SFA and PMOO – even when analyzing larger networks, but with a gain in tightness exceeding 50 % in some cases.

Future Work Directions: Our deep-learning assisted NC framework of Section III is already able to create other heuristics than DeepTMA. For instance, DeepTMA$_n$ with n decompositions suggested per tandem or a heuristic predicting the most computationally efficient decomposition if multiple ones will provide best results. Moreover, it can be further optimized by finding the best number of variables for the graph neural network as well as its bottleneck. Learning from a dataset including feed-forward networks, more complex curve shapes than the simple affine ones [23], or from/for FIFO-multiplexing networks [39] or entirely non-FIFO [40] is already possible, too.

Secondly, our framework is highly extensible. With some minor additions, it can predict if an optional feature of the NC analysis might be beneficial at a certain point. Examples are the alternative output bound formulation of [41] and flow prolongation [42]. Exhaustive flow prolongation is only feasible on single tandems, yet, learning from those can be sufficient as we show in our paper. Another direction is to predict a bound on the necessary domain of curves [30, 43].

REFERENCES

[1] F. Geyer and G. Carle, "Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches," *IEEE Commun. Mag.*, vol. 54, no. 2, pp. 106–112, 2016.

[2] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *Proc. of IEEE Globecom*, 2013.

[3] X. Jin, N. Guan, J. Wang, and P. Zeng, "Analyzing multimode wireless sensor networks using the network calculus," *Journal of Sensors*, vol. 2015, Article ID 851608.

[4] J. W. Guck, A. Van Bemten, and W. Kellerer, "DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 4, pp. 1003–1017, 2017.

[5] S. Bondorf and J. B. Schmitt, "Boosting sensor network calculus by thoroughly bounding cross-traffic," in *Proc. of IEEE INFOCOM*, 2015.

[6] S. Vastag, "Modeling quantitative requirements in SLAs with network calculus," in *Proc. of ICST ValueTools*, 2011.

[7] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. of ACM SoCC*, 2014.

[8] E. J. Rosensweig and J. Kurose, "A network calculus for cache networks," in *Proc. of IEEE INFOCOM*, 2013.

[9] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. of ACM SIGCOMM*, 2015.

[10] A. Charny and J.-Y. Le Boudec, "Delay bounds in a network with aggregate scheduling," in *Proc. of QoFIS*, 2000.

[11] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proc. ACM Meas. Anal. Comput. Syst. (POMACS)*, vol. 1, no. 1, pp. 16:1–16:34, 2017.

[12] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[13] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. of IEEE IJCNN*, 2005.

[14] S. Bondorf and J. B. Schmitt, "The DiscoDNC v2 – a comprehensive tool for deterministic network calculus," in *Proc. of EAI ValueTools*, 2014.

[15] F. Geyer, "Performance evaluation of network topologies using graph-based deep learning," in *Proc. of EAI ValueTools*, 2017.

[16] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *Proc. of GI/ITG MMB*, 2008.

[17] A. Scheffler, M. Fögen, and S. Bondorf, "The deterministic network calculus analysis: Reliability insights and performance improvements," in *Proc. of IEEE CAMAD*, 2018.

[18] J.-Y. Le Boudec, "A theory of traffic regulators for deterministic networks with application to interleaved regulators," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2721–2733, 2018.

[19] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, 2009.

[20] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proc. of ICLR*, 2016.

[21] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. of EMNLP*, 2014.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. of ICLR*, 2018.

[23] A. Bouillard and É. Thierry, "An algorithmic toolbox for network calculus," *Discrete Event Dynamic Systems*, vol. 18, no. 1, 2008.

[24] M. Amrani, L. Lúcio, and A. Bibal, "ML + FV = ♡? A survey on the application of machine learning to formal verification," 2018, arxiv:1806.03600.

[25] H. Al-Zubaidy, J. Liebeherr, and A. Burchard, "A (min, ×) network calculus for multi-hop fading channels," in *Proc. of IEEE INFOCOM*, 2013.

[26] J. Liebeherr, "Duality of the max-plus and min-plus network calculus," *Found. Trends. Network.*, vol. 11, no. 3-4, pp. 139–282, 2017.

[27] M. Boyer and P. Roux, "Embedding network calculus and event stream theory in a common model," in *Proc. of IEEE ETFA*, 2016.

[28] K. Lampka, S. Perathoner, and L. Thiele, "Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems," in *Proc. of ACM EMSOFT*, 2009.

[29] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. of ISCAS*, 2000.

[30] N. Guan and W. Yi, "Finitary real-time calculus: Efficient performance analysis of distributed embedded systems," in *Proc. of IEEE RTSS*, 2013.

[31] A. Bouillard, L. Jouhet, and É. Thierry, "Tight performance bounds in the worst-case analysis of feed-forward networks," in *Proc. of IEEE INFOCOM*, 2010.

[32] S. K. Khangura, M. Fidler, and B. Rosenhahn, "Neural networks for measurement-based bandwidth estimation," in *Proc. of IFIP Networking*, 2018.

[33] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. of NIPS*, 2017.

[34] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. of NIPS*, 2015.

[35] I. Henrion, K. Cranmer, J. Bruna, K. Cho, J. Brehmer, G. Louppe, and G. Rochette, "Neural message passing for jet physics," in *Proc. of the Deep Learning for Physical Sciences Workshop*, 2017.

[36] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," 2018, arxiv:1802.03685.

[37] F. Geyer and G. Carle, "The case for a network calculus heuristic: Using insights from data for tighter bounds," in *Proc. of NetCal*, 2018.

[38] F. Geyer, "DeepComNet: Performance evaluation of network topologies using graph-based deep learning," *Performance Evaluation*, 2018.

[39] A. Bouillard and G. Stea, "Exact worst-case delay in FIFO-multiplexing feed-forward networks," *IEEE/ACM Trans. Net.*, vol. 23, no. 5, pp. 1387–1400, 2015.

[40] J. B. Schmitt, N. Gollan, S. Bondorf, and I. Martinovic, "Pay bursts only once holds for (some) non-FIFO systems," in *Proc. of IEEE INFOCOM*, 2011.

[41] Y. Tang, N. Guan, W. Liu, L. T. X. Phan, and W. Yi, "Revisiting GPC and AND connector in real-time calculus," in *Proc. of IEEE RTSS*, 2017.

[42] S. Bondorf, "Better bounds by worse assumptions – improving network calculus accuracy by adding pessimism to the network model," in *Proc. of IEEE ICC*, 2017.

[43] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan, and W. Yi, "Generalized finitary real-time calculus," in *Proc. of IEEE INFOCOM*, 2017.

A.1.6 On the Robustness of Deep Learning-predicted Contention Models for Network Calculus

This work was published in *Proceedings of the 25th IEEE Symposium on Computers and Communications*, 2020 [64].

On the Robustness of Deep Learning-predicted Contention Models for Network Calculus

Fabien Geyer
Technical University of Munich | Airbus CRT
Munich, Germany

Steffen Bondorf
Faculty of Mathematics, Center of Computer Science
Ruhr University Bochum, Germany

***Abstract*—The network calculus (NC) analysis takes a simple model consisting of a network of schedulers and data flows crossing them. A number of analysis "building blocks" can then be applied to capture the model without imposing pessimistic assumptions like self-contention on tandems of servers. Yet, adding pessimism cannot always be avoided. To compute the best bound on a single flow's end-to-end delay thus boils down to finding the least pessimistic contention models for all tandems of schedulers in the network – and an exhaustive search can easily become a very resource intensive task. The literature proposes a promising solution to this dilemma: a heuristic making use of machine learning (ML) predictions inside the NC analysis.**

While results of this work are promising in terms of delay bound quality and computational effort, there is little to no insight on when a prediction is made or if the trained machine can achieve similarly striking results in networks vastly differing from its training data. In this paper we address these pending questions. We evaluate the influence of the training data and its features on accuracy, impact and scalability. Additionally, we contribute an extension of the method by predicting the best n contention model alternatives in order to achieve increased robustness for its application outside the training data. Our numerical evaluation shows that good accuracy can still be achieved on large networks although we restrict the training to networks that are two orders of magnitude smaller.

I. Introduction

Deterministic bounds on the end-to-end delay are strictly required in many application areas. Prime examples are data networks in avonics and the automotive industry that are shared between multiple distributed x-by-wire applications [1] as well as safety-critical (factory) systems [2, 3, 4, 5].

Network Calculus (NC) is a versatile framework for the derivation of such bounds. The NC literature provides modeling and analysis tooling such that all steps towards derivation of delay bounds can be taken. There exist results on system modeling, ranging from generic behavior like FIFO [6, 7, 8], non-FIFO [9, 10] or unknown [11] to modern technologies such as IEEE Audio/Video Bridging (AVB) and Time-Sensitive Networking (TSN) [12, 13, 14, 15, 5]. Behavior of such queues, schedulers, shapers etc. are modeled as servers that, in turn, are connected to form a network, the so-called server graph [16, 17].

The server graph carries data flows, exactly one of which will be the designated flow of interest (foi) whose delay is bounded by the NC analysis. For this network analysis step, results have been created to capture the modeled system behavior as closely as possible. For example, on tandems of work-conserving servers, neither the worst-case burstiness of the foi nor its cross-flows should impact the delay bound computation more than once – i.e., contention for the forwarding resource should not be assumed more pessimistically by the analysis than actually modeled by the server graph. These two core properties of NC are known as pay bursts only once (PBOO) [11, 6, 18, 19] and pay multiplexing only once (PMOO) [20, 21], respectively. Other such refinements try to reduce the amount of mutually exclusive contention assumptions for multiple flows at shared servers (pay segregation only once, PSOO) [22], paying for multiplexing in ring networks less often (pay multiplexing only at convergence points, PMOC) [23], capping the worst-case burstiness with a server's queue length [24] or using an entirely different alternative to compute bounds on the arrivals of cross-flows [25]. Some of these results are mutually exclusive, e.g., the different arrival bounding method is based on violating the PSOO property.

Capturing these restrictions on realistic worst-case contention in the given model of connected servers correctly does not only help to improve the computed delay bound. It also allows to rank different networks more accurately by not discriminating an alternative that features a design element NC can only consider by pessimistic overapproximation. This allows NC to be used to compare existing network designs to newly proposed ones [2]. However, there is not a single-best NC analysis[1]. In this paper, we focus on networks where there is no knowledge about the multiplexing behavior of flows. This assumption is called arbitrary (or blind) multiplexing. The best delay bound for a flow crossing a cycle-free network of arbitrary multiplexing servers is computed by a specific combination of the properties, the "building blocks", mentioned above. The exhaustive search for this combination has been improved such that it becomes feasible to execute[2] but it still tends to scale superlinearly with the network size [27]. This search-based analysis is called tandem matching analysis (TMA).

Based on TMA, DeepTMA [28] was recently proposed to alleviate this search-induced problem. DeepTMA is a fast heuristic based on deep learning (DL) that replaces the

[1] In this paper, we restrict our presentation to the algebraic analysis methods. The optimization analyses in [26, 8] are indeed best w.r.t. to delay bounds but as shown in [27, 8], they tend to become computationally infeasible.

[2] Moreover, its delay bounds are very close to the optimization approach of [26].

expensive search with a prediction of the best combination of existing results, i.e., the best contention model. While DeepTMA showed promising results towards fast and accurate NC analysis, understanding how predictions are made remains opaque and does not bring insights in the NC analysis or the wider applicability of the method. We aim in this paper to address those drawbacks by evaluating the influence of the dataset used in the training phase, as well as its features, on the eventual prediction accuracy. We also contribute an extension of DeepTMA which is able to generate more than one contention model prediction, leading to an increase of the robustness of the method.

We show that DeepTMA is able to cope with scalability, namely that it can be trained on small networks and being used on much larger networks with low impact on the accuracy. Our numerical evaluation illustrates that the relative error of DeepTMA is still below $1\,\%$ on average when evaluated on networks two orders of magnitude larger than the ones used for training. We also show that training DeepTMA on random networks leads to good applicability on more specific types of networks. Additionally, we give insight into the importance of network features with respect to predicting a contention model. Overall, we first demonstrate DeepTMA's robustness regarding the relation of training set to evaluated network in terms of size and shape. Finally, we evaluate our extension to DeepTMA that proposes multiple alternative contention models. Our evaluation shows that the robustness of DeepTMA can be increased by generating multiple contention models, leading to a decrease of the error with a factor 2.

The remainder of the paper is organized as follows: First, we review related work in Section II. Section III Graph Neural Networks (GNNs) and their combination with Network Calculus. In Section IV ,we present our extension of DeepTMA and the generation of a dataset to learn from. A numerical evaluation of the robustness of DeepTMA is performed in Section V. Finally Section VI concludes our work and gives an outlook.

II. Related work

Research on combining machine learning with formal methods has been found in a variety of applications, e.g., in theorem proving, model-checking or in SAT-SMT problems. In the following, we aim to provide a focused depiction of efforts that are interesting and related to our work. Namely, the performance in networks and Graph Neural Networks (GNNs). A more comprehensive survey on machine learning-assisted formal methods can be found in [29].

GNNs were first introduced in [30, 31], a concept subsequently refined in recent works. Message-passing neural network were introduced in [32], with the goal of unifying various GNN and graph convolutional concepts. [33] formalized graph attention networks, which enables to learn edge weights of a node neighborhood. Finally, [34] introduced the graph networks (GN) framework, a unified formalization of many concepts applied in GNNs.

These concepts were applied to many domains where problems can be modeled as graphs: chemistry with molecule analysis [35, 32], solving the traveling salesman problem [36], prediction of satisfiability of SAT problems [37], or basic logical reasoning tasks and program verification [38]. For computer networks, they have recently been applied to prediction of average queuing delay [39] and different non-NC-based performance evaluations of networks [40, 41, 42]. In the realm of NC, there is surprisingly little work as of yet. Predating DeepTMA [28] we base our work on, there is an effort to predict the delay bound computed by different NC analyses by using GNNs. Each of these analyses only considers a pre-defined contention model whenever there are alternatives for a tandem. The prediction is then used to only execute the most promising analysis [43]. This was developed into DeepTMA that can provide multiple predictions per analysis. Independent efforts aim at predicting delay bounds, too. This work [44, 45] uses supervised learning and benchmarks the predictions against a NC-based analysis. Another similar goal to our work is to provide small yet controllable computation times to make the proposed analysis fit for application in design space exploration.

Regarding assessing the robustness of GNNs, [37, 36] showed that GNNs can be trained on a given set of graphs while being able to extrapolate on other types or much larger graphs. Finally, [46] recently proposed an approach to explain predictions from GNNs, by reducing the input graphs to subgraphs containing a small subset of nodes which are most influential for the prediction.

III. Background: Graph Neural Network for NC

We give a brief overview of the DeepTMA heuristic in this section. We refer the reader to [28] for the full formulation of the method. It is based on the concept of Graph Neural Network (GNN) introduced in [30, 31]. The goal of DeepTMA is to predict the best tandem decompositions, i.e., contention models, to use in TMA. We define networks to be in the NC modeling domain and to consist of servers, crossed by flows. We refer to the model used in GNN as graphs. The main intuition is to transform the networks into graphs. Those graph representations are then used as inputs for a neural network architecture able to process general graphs, which will then predict the tandem decomposition resulting in the best residual service curve. Our approach is illustrated in Figure 1. Since the delay bounds are still computed using the formal network calculus analysis, they inherit their provable correctness.

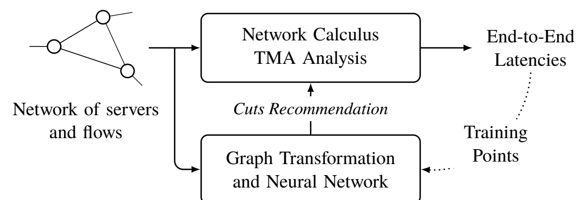


Figure 1: Overview of the proposed approach.

A. Overview of Graph Neural Networks

In this section, we detail the neural network architecture used for training neural networks on graphs, namely the family of architectures based on GNNs [30, 31].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $(v, u) \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and output values for node v. The concept behind GNNs is called *message passing*, where hidden states of nodes $\mathbf{h}_v$ (i.e. vectors of real numbers) are iteratively passed between neighboring nodes.

At each iteration of message passing, each node in the graph aggregates the hidden states of its neighbors, use this aggregate to update its own hidden state, and sends the updated state at the next iteration:

$$\mathbf{h}_v^{(t=0)} = init\left(\mathbf{i}_v\right) \tag{1}$$

$$\mathbf{h}_v^{(t+1)} = aggr\left(\left\{\mathbf{h}_u^{(t)} \;\middle|\; u \in \textsc{Nbr}(v)\right\}\right) \tag{2}$$

with $\mathbf{h}_v^{(t)}$ representing the hidden state of node v at iteration t, $aggr$ a function which aggregates the set of hidden states of the neighboring nodes $\textsc{Nbr}(v)$ of v, and $init$ a function for initializing the hidden states based on the input features. Those hidden states are propagated throughout the graph using multiple iterations of Equation (2) until a fixed point is found. The final hidden state is then used for predicting properties about nodes:

$$\mathbf{o}_v = out\left(\mathbf{h}_v^{(t\to\infty)}\right) \tag{3}$$

with out a function transforming the final hidden state to the target values.

In GNNs, the aggregation of hidden states corresponds to their sum, and the $aggr$ and out functions are feed-forward neural networks (FFNN) such that:

$$\mathbf{h}_v^{(t+1)} = aggr\left(\left\{\mathbf{h}_u^{(t)} \;\middle|\; u \in \textsc{Nbr}(v)\right\}\right) = aggr\left(\sum_{u\in\textsc{Nbr}(v)} \mathbf{h}_u^{(t)}\right) \tag{4}$$

Various extensions of GNNs have been recently proposed in the literature. We selected Gated Graph Neural Networks (GGNN) [38] for implementing DeepTMA, extended with an attention mechanism similar to the one proposed in [33]. This extension implements $aggr$ using a recurrent unit and unrolls Equation (2) for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing.

In order to propagate the hidden states throughout the complete graph, a fixed number of iterations are performed. This extension has been shown to outperform the original formulation of GNNs which require to run the iteration until a fixed point is found. We refer to [34] for additional details on GNNs.

B. Application to TMA

In order to apply the concepts described in Section III-A to a network calculus analysis, we a simple NC network (servers and a data flow) into a graph. Figure 2 illustrates this transformation.

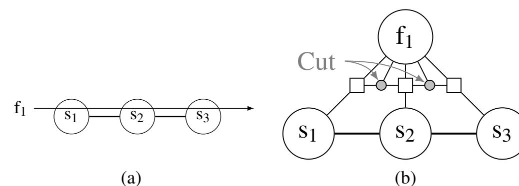


Figure 2: Graph representation of a sample tandem network.

Each server is represented as a node in the graph, with edges corresponding to the network's links. Each flow is represented as a node. The path taken by a flow in this graph, is encoded using edges which connect the flow to the servers it traverses. Since those edges do not encode the order in which those servers are traversed, so-called *path ordering* nodes are added to edges between the flow node and the traversed server nodes. This property is especially important in the TMA since the order, and hence position of cuts, has an impact on dependency structures. In order to represent these TMA cuts, each potential cut between pairs of servers on the path traversed by the flow is represented as a node. This cut node is connected via edges to the flow and to the pair of servers it is associated to.

In addition to a categorical encoding of the node type (i.e., server, flow, path ordering or cut), the input features of each node in the graph need to comprise some NC definitions. A comprehensive treatment of NC can be found in [11]. TMA and DeepTMA are described in [27] and [28], respectively. NC resource models rely on non-negative, wide-sense increasing functions

$$\mathcal{F}_0 = \left\{ f : \mathbb{R} \to \mathbb{R}_\infty^+ \;\middle|\; f(0) = 0,\ \forall s \le t : f(s) \le f(t) \right\},$$

where $\mathbb{R}_\infty^+ := [0, +\infty) \cup \{+\infty\}$. These functions pass through the origin such that there are no instantaneous data occurrences in the functions (in absolute time t) that cumulatively count input and output data, $A(t)$ and $A'(t)$. The server graph crossed by flows, in short network, is annotated with different NC-functions of $\mathcal{F}_0$; two kinds of so-called curves, each one bounding a relevant property in interval time.

Definition 1 (Arrival Curve): Let the data arrivals of a flow over time be characterized by function $A(t) \in \mathcal{F}_0$, where $t \in \mathbb{R}_\infty^+$. An arrival curve $\alpha(d) \in \mathcal{F}_0$ for $A(t)$ must then fulfill

$$\forall t\, \forall d,\ 0 \le d \le t : A(t) - A(t-d) \le \alpha(d),$$

i.e., it must bound the flow's data arrivals in any duration d.

*Definition 2 (*Strict *Service Curve): If, during any period with backlogged data of duration d, a scheduler, queue, etc. with input function A guarantees an output of at least $\beta(d) \in \mathcal{F}_0$, then it is said to offer a* strict *service curve β.*

These are incorporated as follows:

- For each server s, parameters of its rate-latency service curve are used where $\beta_s(d) = \max\{0, rate_s \cdot d - latency_s\}$: $[rate_s, latency_s]$
- For each flow f, parameters of its token bucket arrival curve are used where $\alpha_f(d) =$

$\{rate_f \cdot d + burst_f\}_{\{d>0\}}$ (i.e., $\alpha_f(d) = 0$ for $d \leq 0$): $[rate_f, burst_f]$

- For each path ordering p, the hop count is encoded as a categorical one-hot vector: $[hop = 1, \dots, hop = n]$
- Finally, cut nodes do not have input features

Note that in case more complex arrival or service curve shapes than affine curves [47] are studied, those input features can be extended to represent the additional curve parameters. Last, note that edges have no features in this graph encoding.

Since the goal of DeepTMA is to predict which tandem decomposition will result in the tightest bound, only the nodes presenting cuts have output features. This problem is formulated as a classification problem, namely each cut node has to be classified in two classes: perform a cut between the pair of servers it is connected to or not: $[cut, \overline{cut}]$. The overall prediction to be fed back, i.e., the selection of one out of TMA's potential decompositions for a given foi's path, is defined by the set of all cut classifications for this path.

IV. Increasing the Robustness of DeepTMA

A. DeepTMA$_n$: Generate multiple tandem decompositions

Given a foi and a potential cut location, the output of the neural network is a probability of cutting. This probability is generated by the neural network using the softmax function after its last layer. In case a single tandem decomposition has to be generated, the decision of cutting is made using a threshold of $50\,\%$.

Those cut probabilities may also be used in order to generate multiple tandem decompositions as illustrated in Algorithm 1. In case the number of tandem decompositions is lower than the number of requested decompositions, we simply return all combinations of cuts. Otherwise, we sample the distribution of cuts in order to generate the decompositions. We label this extension of DeepTMA as DeepTMA$_n$, with n the number of tandem decompositions generated.

Algorithm 1 Generation of n tandem decompositions for a flow traversing $L+1$ servers.

if $n \leq L^2$ **then return** all combinations of cuts
else
  for all $i := 1$ **to** n **do**
    $v \leftarrow [c_1, \dots, c_L] \sim \mathcal{U}(0,1)^L$
    $\text{cuts}_i \leftarrow \mathbb{I}\left(v \leq \left[\Pr(cut^{GNN}_{foi,1}), \dots, \Pr(cut^{GNN}_{foi,L})\right]\right)$
    ($\mathbb{I}$ is the indicator function)
  return $\{\text{cuts}_1, \dots, \text{cuts}_n\}$

B. Dataset generation

In order to train our neural network architecture, we randomly generated a set of topologies according to three different random topology generators: *a)* tandems or daisy-chains, *b)* trees and *c)* random server graphs following the $G(n,p)$ Erdős–Rényi model [48]. For each created server, a rate latency service curve was generated with uniformly random rate and latency parameters. A random number of flows with random source and sink servers was added. Note that in our topologies, there cannot be cyclic dependency between the flows. For each flow, a token bucket arrival curve was generated with uniformly random burst and rate parameters. All curve parameters were normalized to the $(0,1]$ interval.

In total, 172 374 different networks were generated, with a total of more than 13 million flows, and close to 260 million tandem decompositions. Half of the networks were used for training the neural network, while the other half was used for the evaluation presented later in Section V. Table I summarizes different statistics about the generated dataset. The dataset is will be available online to reproduce our learning results. Note that compared to the original dataset used for training DeepTMA [28], this dataset contains larger networks.

Parameter	Min	Max	Mean	Median
# of servers	2	41	14.6	12
# of flows	3	203	101.2	100
# of tandem combinations	2	197 196	1508.5	384
# of nodes in analyzed graph	10	2093	545.2	504
# of tandem combination per flow	2	65 536	19.4	4
# of flows per server	1	173	18.1	10

Table I: Statistics about the randomly generated dataset.

Additionally to this dataset, we also evaluate our approach on the set of networks used in [27]. Table II summarizes different statistics about the generated dataset. Compared to dataset used for training, this additional set of networks up to two order of magnitude larger in term of number of servers and flows per network. This property will be used in Section V in order to evaluate if our approach is able to scale to such larger networks, both in term of accuracy and execution time.

Parameter	Min	Max	Mean	Median
# of servers	38	3626	863.0	693
# of flows	152	14 504	3452.0	2772
# of tandem combinations	2418	121 860	24 777.6	18 869
# of nodes in analyzed graph	1358	113 162	25 137.7	19 518
# of tandem combination per flow	2	512	7.3	8
# of flows per server	1	467	16.4	12

Table II: Statistics about the set of networks from [27].

V. Numerical evaluation

We evaluate in this section our extensions of DeepTMA as well as its robustness and scalability. Via a numerical evaluation, we illustrate the tightness and execution time of DeepTMA and highlight its usability for practical use-cases. Details on the datasets used for this evaluation were presented in Section IV-B.

In order to numerically evaluate and compare DeepTMA against TMA, we selected the relative error metric as our main metric for the rest of this evaluation. This metric measures the relative difference of DeepTMA against TMA with respect to the end-to-end delay bound, and is defined for flow f_i as:

$$RelErr_{f_i} = (Delay^{DeepTMA}_{f_i} - Delay^{TMA}_{f_i}) / Delay^{TMA}_{f_i} \quad (5)$$

A. *Impact of training network sizes on error*

Following the previous evaluation, we investigate here the scalability of DeepTMA by evaluating the impact of the training dataset on the accuracy of the method. We trained here two additional instances of the deep-learning part of DeepTMA. Each has a different restriction on the maximum amount of flows in the networks to be included in the training set, namely 50 and 100.

Figure 3 illustrates the error of those additional instances of DeepTMA compared to the one trained on the full dataset. There is an evident trend that a smaller training set size as imposed by our restriction generally leads to an increasing relative error. However, there is one exception at path length 17. This illustrates that the "quality" of the training set can be more important than its size. We provide a closer look at network type and features as potential impact factors for the training set quality in the Sections V-B and V-C.

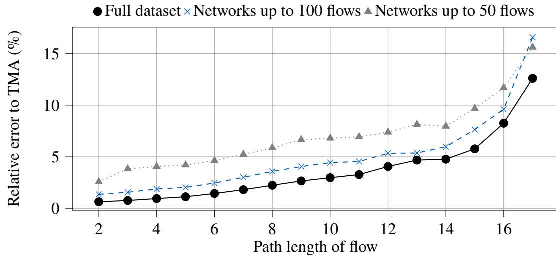


Figure 3: Influence of training size on relative error of DeepTMA

Similarly, Figure 4 illustrates the impact of training dataset on the set of networks from [27]. The difference between the different variants of DeepTMA is minimal except on the large networks. This indicates that DeepTMA is still able to scale, even when trained on much smaller networks. Moreover, we can see small datasets outperforming the full set again in some cases.

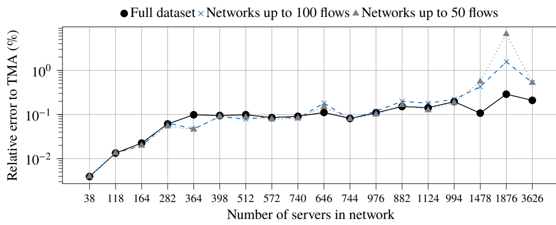


Figure 4: Influence of training size on the set of networks from [27]

B. *Influence of network type used for training*

We examine in this section the impact of the network types used for training DeepTMA on its accuracy. As explained in Section IV-B, three different types of networks were generated, namely *a)* tandems, *b)* trees and *c)* random server graphs based on the $G(n, p)$ Erdős–Rényi model.

We evaluate the ability of DeepTMA to extrapolate on other networks by training three different variants for DeepTMA, each on one type of networks. Results are presented in Figure 5. Compared DeepTMA trained on the full dataset, training only on tandem or tree networks leads to good ability at extrapolating on other types of networks. Surprisingly, tree network-based training dataset is outperformed by the tandem-based one that, in turn, is very competitive with the random server graphs.

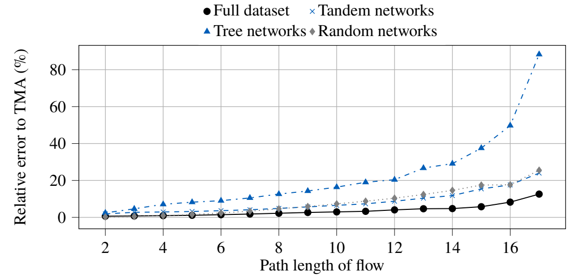


Figure 5: Influence of network types used for training on relative error of DeepTMA

C. *Importance of features and locality*

In order to better understand the importance of the input features used in DeepTMA, we assess each feature's importance following the permutation-based importance measure [49, 50]. For each input feature presented in Section III-B, we randomize it by randomly permuting its values in the training set, and assess the impact it has on the accuracy of the predictions. We define the importance metric as:

$$Importance(Feature) = \frac{1}{|\mathcal{F}|} \sum_{f_i \in \mathcal{F}} \left(RelGap_{f_i}^{Feature} - RelGap_{f_i}^{Baseline} \right) \quad (6)$$

with the baseline corresponding to DeepTMA without any feature permutation. With this evaluation, we assess how much the GNN model relies on a given feature of interest for making its prediction.

Features importance are presented in Figure 6(above). The service rate of the servers in the network have the largest influence on the final decision of cutting. Such behavior confirms an existing result of NC, which is known to be sensitive to service rate. The remaining features appear to have less importance on the cut prediction. Interesting from the NC perspective is the observation that the order of servers (PathOrder) has a percental importance two orders of magnitude lower than the service rate. In combination, these two features constitute the very reason for TMA (and optimization-based analyses, see [51]) to outperform the previous NC analyses.

We also assess the importance of other flows and other servers on a cut. We perform this by assessing the number of iterations of message passing (i.e. Equation (2)) and the

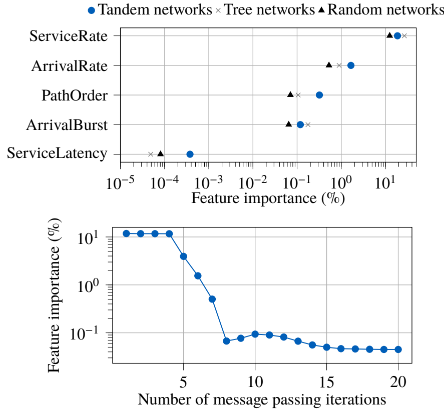
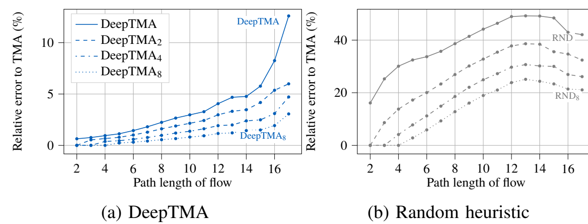


Figure 6: (above) Feature importance for DeepTMA and (below) impact of number of GNN message passing iterations

impact it has on the relative error. As for feature importance, we compare the results according to Equation (6). Results are presented in Figure 6(below). The first 4 loop iterations appear to have the largest influence on the cut decision, meaning that the cut decision is mainly based on information from servers close to the cut. We notice that the importance drops sharply after 5 iterations, and converges after 15 iterations. This indicates that servers and flows farther away from the cut decision are less relevant to the cut decision – an insight to potential further improvement of DeepTMA's tradeoff between computational effort and relative error.

D. Evaluation of DeepTMA$_n$

We now start focusing on actively improving robustness and evaluate our extension of DeepTMA defined in Section IV-A. It enables DeepTMA to generate more than one tandem decompositions. Results are presented in Figure 7(a), where the subscript n denotes the number of tandem decompositions generated by DeepTMA$_n$. To benchmark DeepTMA$_n$, we depict the performance of a random heuristic in Figure 7(b).



(a) DeepTMA (b) Random heuristic

Figure 7: Evaluation of DeepTMA$_n$ and random heuristic

As expected, the generation of more than one tandem decomposition results in a decrease of the error. Already with DeepTMA$_2$, the error is reduced by a factor of 2 on the larger networks. This illustrates that the robustness of DeepTMA can be increased by using Algorithm 1, even at the smallest additional computational cost of going to DeepTMA$_2$. The random heuristic performs considerably worse in all aspects evaluated above.

E. Scalability on large networks

We evaluate in this section the robustness of DeepTMA and DeepTMA$_n$ with respect to scalability. The networks from [27] are evaluated here, since those networks are almost two orders of magnitude larger than the networks used for training the GNN used in DeepTMA, as illustrated in Tables I and II.

Figure 8 illustrates the relative error of DeepTMA and DeepTMA$_n$ compared to a random heuristic which selects the tandem decompositions randomly. The family of DeepTMAs achieve relative errors that are two orders of magnitudes smaller than the random heuristics, resulting in better end-to-end delay bound accuracy w.r.t. the exhaustive TMA.

Although DeepTMA wasn't trained on such large networks, the relative error still stays below $0.3\,\%$ even on the larger networks. DeepTMA$_8$ is even able to reach relative errors below $0.02\,\%$, indicating a good ability to scale.

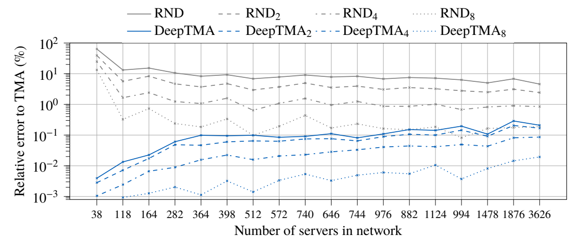


Figure 8: Evaluation of DeepTMA$_n$ on the set of networks from [27]

VI. Conclusion

We contributed in this paper an extension of DeepTMA for generating multiple tandem decomposition predictions and a comprehensive assessment of its robustness in term of scalability and impact of training data on its accuracy. We also provided some insights on which feature is important for making a prediction.

Via a numerical evaluation we showed that DeepTMA can be trained on small networks and still provide good accuracy on much larger networks, up to two order of magnitude larger in term of number of servers and flows. We also showed that the network type used for training can have a large impact on the accuracy of the prediction made by the GNN. Nevertheless, we showed that training DeepTMA on randomly generated networks can still lead to good accuracy, suggesting that tailoring the training data to more realistic use-cases might not be necessary for application on real networks.

Those new results indicate that DeepTMA is able to generalize tandem decomposition rules from small random networks which can also be applied on larger networks, at a low execution time cost. Finally we also proposed an extension of DeepTMA which is able to generate multiple predictions, decreasing the prediction error by a factor of two.

References

[1] F. Geyer and G. Carle, "Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches," *IEEE Commun. Mag.*, vol. 54, no. 2, pp. 106–112, 2016.

[2] A. Amari, A. Mifdaoui, F. Frances, and J. Lacan, "Worst-case timing analysis of AeroRing – a full duplex ethernet ring for safety-critical avionics," in *Proc. of IEEE WFCS*, 2016.

[3] A. Finzi, A. Mifdaoui, F. Frances, and E. Lochin, "Incorporating TSN/BLS in AFDX for mixed-criticality applications: Model and timing analysis," in *Proc. of IEEE WFCS*, 2018.

[4] A. Finzi and S. S. Craciunas, "Integration of SMT-based scheduling with rc network calculus analysis in TTEthernet networks," in *Proc. of IEEE ETFA*, 2019.

[5] J. Zhang, L. Chen, T. Wang, and X. Wang, "Analysis of TSN for industrial automation based on network calculus," in *Proc. of IEEE ETFA*, 2019.

[6] M. Fidler, "Extending the network calculus pay bursts only once principle to aggregate scheduling," in *Proc. of QoS-IP*, 2003.

[7] L. Bisti, L. Lenzini, E. Mingozzi, and G. Stea, "Numerical analysis of worst-case end-to-end delay bounds in fifo tandem networks," *Real-Time Syst.*, 2012.

[8] A. Bouillard and G. Stea, "Exact worst-case delay in FIFO-multiplexing feed-forward networks," *IEEE/ACM Trans. Net.*, vol. 23, no. 5, pp. 1387–1400, 2015.

[9] G. Rizzo and J.-Y. Le Boudec, ""pay bursts only once" does not hold for non-FIFO guaranteed rate nodes," *Perform. Eval.*, vol. 62, no. 1-4, 2005.

[10] J. B. Schmitt, N. Gollan, S. Bondorf, and I. Martinovic, "Pay bursts only once holds for (some) non-FIFO systems," in *Proc. of IEEE INFOCOM*, 2011.

[11] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[12] R. Queck, "Analysis of ethernet avb for automotive networks using network calculus," in *Proc. of IEEE ICVES*, 2012.

[13] J. A. R. De Azua and M. Boyer, "Complete modelling of avb in network calculus framework," in *Proc. of RTNS*, 2014.

[14] J.-Y. Le Boudec, "A theory of traffic regulators for deterministic networks with application to interleaved regulators," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2721–2733, 2018.

[15] H. Daigmorte, M. Boyer, and L. Zhao, "Modelling in network calculus a TSN architecture mixing time-triggered, credit based shaper and best-effort queues," 2018, working paper or preprint.

[16] S. Bondorf and J. B. Schmitt, "The DiscoDNC v2 – a comprehensive tool for deterministic network calculus," in *Proc. of EAI ValueTools*, 2014.

[17] B. Cattelan and S. Bondorf, "Iterative design space exploration for networks requiring performance guarantees," in *Proc. of IEEE/AIAA DASC*, 2017.

[18] G. Chen, K. Huang, C. Buckl, and A. Knoll, "Applying pay-burst-only-once principle for periodic power management in hard real-time pipelined multiprocessor systems," *ACM Trans. Des. Autom. Electron. Syst.*, 2015.

[19] Y. Tang, Y. Jiang, X. Jiang, and N. Guan, "Pay-burst-only-once in real-time calculus," in *Proc. of IEEE RTCSA*, 2019.

[20] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *Proc. of GI/ITG MMB*, 2008.

[21] A. Bouillard, B. Gaujal, S. Lagrange, and É. Thierry, "Optimal routing for end-to-end guarantees using network calculus," *Performance Evaluation*, 2015.

[22] S. Bondorf and J. B. Schmitt, "Should network calculus relocate? an assessment of current algebraic and optimization-based analyses," in *Proc. of QEST*, 2016.

[23] A. Amari and A. Mifdaoui, "Worst-case timing analysis of ring networks with cyclic dependencies using network calculus," in *Proc. of IEEE RTCSA*, 2016.

[24] S. Bondorf and J. B. Schmitt, "Improving cross-traffic bounds in feed-forward networks – there is a job for everyone," in *Proc. of GI/ITG MMB & DFT*, 2016.

[25] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Catching corner cases in network calculus – flow segregation can improve accuracy," in *Proc. of GI/ITG MMB*, 2018.

[26] A. Bouillard, L. Jouhet, and É. Thierry, "Tight performance bounds in the worst-case analysis of feed-forward networks," in *Proc. of IEEE INFOCOM*, 2010.

[27] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proc. ACM Meas. Anal. Comput. Syst. (POMACS)*, vol. 1, no. 1, pp. 16:1–16:34, 2017.

[28] F. Geyer and S. Bondorf, "DeepTMA: Predicting effective contention models for network calculus using graph neural networks," in *Proc. of INFOCOM*, 2019.

[29] M. Amrani, L. Lúcio, and A. Bibal, "ML + FV = ♡? A survey on the application of machine learning to formal verification," 2018, arxiv:1806.03600.

[30] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. of IEEE IJCNN*, 2005.

[31] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, 2009.

[32] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. of NIPS*, 2017.

[33] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. of ICLR*, 2018.

[34] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018, arxiv:1806.01261.

[35] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. of NIPS*, 2015.

[36] M. Prates, P. H. C. Avelar, H. Lemos, L. C. Lamb, and M. Y. Vardi, "Learning to solve NP-complete problems: A graph neural network for decision TSP," *Proc. of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4731–4738, 2019.

[37] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," 2018, arxiv:1802.03685.

[38] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proc. of ICLR*, 2016.

[39] K. Rusek and P. Cholda, "Message-passing neural networks learn little's law," *IEEE Communications Letters*, 2018.

[40] F. Geyer, "Performance evaluation of network topologies using graph-based deep learning," in *Proc. of EAI ValueTools*, 2017.

[41] ——, "DeepComNet: Performance evaluation of network topologies using graph-based deep learning," *Performance Evaluation*, 2018.

[42] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN," 2019.

[43] F. Geyer and G. Carle, "The case for a network calculus heuristic: Using insights from data for tighter bounds," in *Proc. of NetCal*, 2018.

[44] T. L. Mai, N. Navet, and J. Migge, "A hybrid machine learning and schedulability analysis method for the verification of TSN networks," in *Proc. of IEEE WFCS*, 2019.

[45] ——, "On the use of supervised machine learning for assessing schedulability: Application to ethernet TSN," in *Proc. of RTNS*, 2019.

[46] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Proc. of NeurIPS*, 2019.

[47] A. Bouillard and É. Thierry, "An algorithmic toolbox for network calculus," *Discrete Event Dynamic Systems*, vol. 18, no. 1, 2008.

[48] P. Erdős and A. Rényi, "On random graphs. i," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[49] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[50] A. Fisher, C. Rudin, and F. Dominici, "Model class reliance: Variable importance measures for any machine learning model class, from the "rashomon" perspective," 2018.

[51] J. B. Schmitt, F. A. Zdarsky, and M. Fidler, "Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch…," in *Proc. of IEEE INFOCOM*, 2008.

A.1.7 Graph-based Deep Learning for Fast and Tight Network Calculus Analyses

This work was published in *IEEE Transactions on Network Science and Engineering*, 2021 [65].

IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING, VOL. 8, NO. 1, JANUARY-MARCH 2021 75

Graph-Based Deep Learning for Fast and Tight Network Calculus Analyses

Fabien Geyer and Steffen Bondorf

***Abstract*—Network Calculus (NC) computes end-to-end delay bounds for individual data flows in networks of aggregate schedulers. It searches for the best model bounding resource contention between these flows at each scheduler. The literature proposes different analyses to consider realistic behavior of networked system such as multiplexing, and contention between flows in consecutive queues even though there is no knowledge on the multiplexing discipline employed by the crossed systems (arbitrary multiplexing property). Bounding delays in entire feed-forward networks needs to keep track of such behavior. Moreover, not a single of the existing fast NC heuristics that are based on an algebraic analysis is strictly best. An exhaustive search for the best combination of analyses, i.e., contention modeling, was proposed with the Tandem Matching Analysis (TMA). Additional measures made it scale best among the NC analyses, yet bounding delays may still require several hours of computation time. In this paper, we demonstrate the ability to couple graph-based neural networks with NC by extending TMA with a prediction mechanism replacing the exhaustive search. We propose a framework that learns from NC's TMA, predicts best contention models, and feeds them back to TMA where the according NC computations are executed. We achieve provably valid bounds that are very competitive with the exhaustive TMA. We observe a maximum relative error to TMA below 12%, while execution times remain nearly constant, and outperform TMA in differently sized networks by several orders of magnitude.**

***Index Terms*—Deep Learning, Network Calculus.**

I. Introduction

Deterministic performance bounds have seen many applications in modern systems and a wide range of network calculus-based solutions have been proposed. Network Calculus (NC) can be applied to ensure deadlines in networks for x-by-wire applications [1] as well as SDN-enabled networks [2], for safety-critical production systems [3], or both of these [4]. Moreover, NC solutions have been proposed for highly dynamic environments. E.g., admission control in self-modeling sensor networks [5] or systems providing customers with service level agreements [6] for, among others, storage access [7]. Other recent examples where dynamic events may often cause changes are cache networks [8] and cloud computing [9]. These areas benefit from fast computations of tight performance bounds. The literature provides one-shot analyses for topology-agnostic bounds [10] or bounds that hold for the specification's worst case [4]. Yet, these attempts are ultimately paid for with wasted resources. Our approach aims for highest quality of bounds as well as providing a fast analysis that considers all details of the analyzed network.[1]

A. Problem Overview

In network calculus, a network needs to be modeled by servers (e.g., queues or packet schedulers) whose forwarding capabilities for can be lower bounded. They guarantee an output for their aggregate input of data. Individual data flows traverse sequences of servers where they compete for the forwarding resources with other flows. We do not assume any knowledge about the way data of distinct flows is multiplexed into shared queues at common servers. In NC, this is called arbitrary (or blind) multiplexing. We only assume that the FIFO order of data within individual flows is retained when being multiplexed and forwarded. The data put into a network by a flow is upper bounded in the NC model. This model enables NC to compute deterministic delay bounds.

The NC analysis will compute a bound on an individual flow's end-to-end delay based on such a model. The analyzed flow is commonly known as flow of interest (foi). Under the assumption that no knowledge about the multiplexing of flows is available, the NC analysis must find an internal model of flow contention that

1) bounds the realistic system's worst-case behavior in the foi's point of view without adding too much pessimism and
2) can be solved with the capabilities of the available NC analyses.

The set of available analyses has been steadily extended in order to capture different features of the modeled network

Manuscript received April 3, 2020; revised August 5, 2020 and September 16, 2020; accepted September 18, 2020. Date of publication September 22, 2020; date of current version March 17, 2021. The work of Fabien Geyer's was supported by the German-French Academy for the Industry of the Future. The work of Steffen Bondorf's was supported in part by a Carl-Zeiss Foundation Fellowship, carried out in the Distributed Computer Systems Lab at TU Kaiserslautern, Germany, and in part by the Network Research Laboratory at the University of Toronto, Canada, and in part by the tenure of an ERCIM 'Alain Bensoussan' Fellowship Programme in the Department of Information Security and Communication Technology at NTNU Trondheim, Norway, and in part by the Faculty of Mathematics' Center of Computer Science at Ruhr University Bochum, Germany. Recommended for acceptance by Dr. Shiwen Mao. *(Corresponding author: Fabien Geyer.)*

Fabien Geyer is with the Airbus Central Research and Technology, Technical University of Munich, 85748 Garching by Munich, Germany (e-mail: fabien.geyer@tum.de).

Steffen Bondorf is with the Faculty of Mathematics, Ruhr University Bochum, 44801 Bochum, Germany (e-mail: steffen.bondorf@rub.de).

Digital Object Identifier 10.1109/TNSE.2020.3025806

[1] A first version of this work was presented at the 2019 IEEE International Conference on Computer Communications (INFOCOM) [11]

2327-4697 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

for tightening the derived delay bounds [12]–[15]. These alternatives are all proven to result in valid delay bounds for the foi. But among the analyses that can be derived in an algebraic fashion, there is not a single-best one that expresses the realistic worst-case contention model without adding pessimism in some other regard, not even on a tandem of two servers crossed by two flows [16]. Most importantly for our work, we inherit the restriction to a specific shape of curves bounding arrivals and forwarding from [15]: arrivals must be upper-bounded by the minimum of several token-bucket constraints and forwarding service must be lower-bounded by the maximum of several rate-latency constraints. This is, however, a constraint commonly found in a multitude of other NC analyses, too [16]–[18].

All the worse for NC, such an algebraic analysis needs to bound the impact of resource contention by transforming the flows' bounding curves between their respective source to the location of contention with the analyzed foi. Curve transformations thus require to backtrack all cross-flows, either in aggregate or separated by worst-case priority assumptions. Different contention models require different flow aggregation/separation assumptions and the resulting structures expressing dependencies of algebraic NC operations become unique. I.e., they all need to be computed.

It was shown that it is possible to exhaustively derive all dependency structures and rank each contention model on each tandem occurring in a network analysis. This is known as the Tandem Matching Analysis (TMA) [19]. It achieves high degrees of delay bound tightness by enumerating all contention models upstream from the foi. Thus, the best model for a downstream location and flow can be found. TMA provides a recursive algorithm whose execution time can exceed several hours, e.g., when analyzing networks with >1000 servers and four times as many flows.

In this article, we present the deep-learning assisted TMA, DeepTMA, that predicts the best contention model with high efficacy, resulting in a high degree of delay bound tightness. Single backtrackings have been attempted before [14], [15], yet, we are the first to achieve considerably faster execution times than TMA without considerably compromising on delay bound tightness.

B. Contributions

While we focus our evaluations on the novel DeepTMA heuristic for NC's TMA, we contribute an entire underlying framework that combines the theories of NC [14] and a graph-based deep learning, namely Graph Neural Networks (GNNs) [20], as well as two of their tools [21], [22]. We assume here feed-forward networks with rate-latency and arbitrary-multiplexing servers, and rate-latency constrained flows, but our approach may be extend to more complex use-cases. DeepTMA achieves the following properties:

Deterministic bounds: We learn from NC and feed predictions back to NC. We predict the best choices for decisions made during the TMA analyses. NC stays in control and guarantees provably correct bounds.

Our framework does not learn to predict a delay bound but it predicts the most important decisions within the TMA analysis, the contention models. Compared to directly predicting a flow's delay bound, our approach always guarantees for a valid worst-case bound as we continue to apply the proven NC operations in their valid orders.

Fast execution times and high tightness: Recent work [23] about the benefit of technical upscaling showed that TMA cannot be parallelized easily and a speedup of only one order of magnitude was observed. We provide an advancement that improves the execution times of the analysis by multiple order of magnitude.

Limited impact of mismatches between training and application: Naturally, we only train our machine learning part once before using its predictions in DeepTMA. While we chose a reasonably large range of parameters for the involved curve descriptions to learn from, our dataset needs to be restricted in some dimensions. Immediately noticeable is the type of network topologies. We use tandem, sink-tree and random networks for training. An evalutation of the original DeepTMA's performance when applied to other topologies is presented in [24], with a short excerpt in this article.

During the NC network analysis, only one delay bound will be computed – the one for the analyzed flow. The remaining computational effort stems from so-called arrival bounding, the computation of bounds on flow (aggregate) arrivals inside the network. The original DeepTMA was only trained for and applied to minimizing the one delay bound. In this article, we provide an evolution of DeepTMA for arrival bounding while preventing the instantiation and integration of a second, differently trained neural network.

C. Outline

The remainder of the article is organized as follows: Section II presents the related work on our research direction for network calculus and graph neural networks. Section III presents the theory behind our approach in more detail and Section IV presents our theoretical contribution on combining both areas. In Section V we present the combination of tools as well as the generation of a dataset to learn from. Section VI provides new machine learning-based NC heuristics to benchmark DeepTMA against. These numerical benchmarks are presented in Section VII, followed by observations about the deep learning-based NC heuristics in Section VIII. Section IX concludes our work.

II. Related Work

A recent survey [25] about existing applications of machine learning to formal verification shows that this combination can accelerate formal methods, e.g., theorem proving, model-checking, Boolean satisfiability problems (SAT) or satisfiability modulo theories (SMT) problems. As we show, NC has been combined with other methods, too. So have GNNs with formal verification. Yet, we are the first to combine both TMA

and GNN into a framework for deterministic performance analysis.

A. NC Combined With Other Methodologies

The (min,+)-algebraic NC provides deterministic modeling and analysis techniques. It has seen various efforts to extend NC's capabilities. For instance, the underlying (min,+) algebra can be exchanged for (min,$\times$) for fading channel analysis [26] or for (max,+) to better fit discrete event systems [27]. Moreover, a common model for NC and event stream theory has been developed [28] and state-based system modeling can be integrated by pairing NC with timed automata [29].

Stochastic extensions to NC were proposed early to deal with, e.g., traffic arrivals following a distribution that cannot be bounded deterministically by an arrival curve. For instance, Boole or martingale inequalities can be applied [30]–[32]. This branch of NC was also extended to include statistics and statistical uncertainty to obtain stochastic results [33], [34].

NC has been used to describe component models commonly found in real-time systems [35]. Delay bounds can then be derived from a combination of component characteristics and the network calculus model. For example, knowledge about the busy period of a greedy processing component has been used to speed up NC computations [36]–[38].

An optimization formulation has been derived from the NC model that computes tight bounds in networks without assumptions on the multiplexing of flows [17]. It first derives the dependencies between busy periods of servers in order to partially order the mutual impact of flows. The tight analysis requires to expand this order to all compatible total orders. There are several algorithms to solve this challenge. As shown in [19], the resulting amount of total orders and therefore linear programs (LP) to solve can quickly becoming prohibitive. [17] proposes a heuristic that skips the expansion step and still derives valid bounds. Its computational demand was numerically evaluated in [19].

Recent works use machine learning to estimate service curves from measurements [39] or to derive traffic characteristics for performing dynamic resource provisioning [40]. In contrast to our work, this interfacing via service curves cannot compute provably correct bounds on the worst-case flow delays due to uncontrollable uncertainties introduced by measurements and machine learning.

B. Deep Learning for Graphs and Formal Verification

GNNs were first introduced in [20], [41], a concept subsequently refined in recent works. Gated Graph Neural Networks (GGNNs) [42] extended this architecture with modern practices by using Gated Recurrent Unit (GRU) memory units [43]. Message-passing neural network were introduced in [44], with the goal of unifying various GNN and graph convolutional concepts. [45] formalized graph attention networks, which enables to learn edge weights of a node neighborhood. Finally, [46] introduced the graph networks (GN) framework, a unified formalization of many concepts applied in GNNs.

These concepts were applied to many domains where problems can be modeled as graphs: chemistry with molecule analysis [44], [47], solving the traveling salesman problem [48], prediction of satisfiability of SAT problems [49], or basic logical reasoning tasks and program verification [42]. For computer networks, they have recently been applied to prediction of average queuing delay [50] and different non-NC-based performance evaluations of networks [22], [51]–[53]. In the realm of NC, there is surprisingly little work as of yet. Predating DeepTMA [11] we base our work on, there is an effort to predict the delay bound computed by different NC analyses by using GNNs. Each of these analyses only considers a pre-defined contention model whenever there are alternatives for a tandem. The prediction is then used to only execute the most promising analysis [54].

III. Background

A. Overview of Graph Neural Networks

In this section, we detail the neural network architecture used for training neural networks on graphs, namely the family of architectures based on GNNs [20], [41].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $(v, u) \in \mathcal{E}$. Let $\mathbf{i}_v \in \mathbb{R}^n$ and $\mathbf{o}_v \in R^m$ represent respectively the input features (e.g. node type, service or arrival curve parameters) and output values for node v (e.g. decision for the NC analysis). The concept behind GNNs is called *message passing*, where so-called hidden representations of nodes $\mathbf{h}_v \in \mathbb{R}^k$ are iteratively passed between neighboring nodes. Those hidden representations are propagated throughout the graph using multiple iterations until a fixed point is found or after a fixed number of iterations. The final hidden representation is then used for predicting properties about nodes. This concept can be formalized as:

$$\mathbf{h}_v^{(t)} = aggr\Big(\Big\{\mathbf{h}_u^{(t-1)} \mid u \in \textsc{Nbr}(v)\Big\}\Big) \tag{1}$$

$$\mathbf{o}_v = out\Big(\mathbf{h}_v^{(t\to\infty)}\Big) \tag{2}$$

$$\mathbf{h}_v^{(t=0)} = init(\mathbf{i}_v) \tag{3}$$

with $\mathbf{h}_v^{(t)}$ representing the hidden representation of node v at iteration t, $aggr$ a function which aggregates the set of hidden representations of the neighboring nodes $\textsc{Nbr}(v)$ of v, out a function transforming the final hidden representation to the target values, and $init$ a function for initializing the hidden representations based on the input features.

The concrete implementations of the $aggr$ and out functions are feed-forward neural networks (FFNN), with the addition that $aggr$ is the sum of per-edge terms [41], such that:

$$\mathbf{h}_v^{(t)} = aggr\Big(\Big\{\mathbf{h}_{\textsc{Nbr}(v)}^{(t-1)}\Big\}\Big) = f\left(\sum_{u\in\textsc{Nbr}(v)} \mathbf{h}_u^{(t-1)}\right) \tag{4}$$

with f a FFNN. For $init$, a one-layer FFNN is used to fit the input features to the dimensions of the hidden representations.

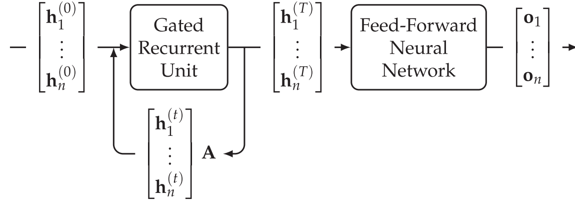


Fig. 1. Gated Graph Neural Network architecture.

Gated Graph Neural Networks (GGNN) [42] were recently proposed as an extension of GNNs to improve their training. This extension implements f using a memory unit called Gated Recurrent Unit (GRU) [43] and unrolls Equation 1 for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing. The neural network architecture is illustrated in Figure 1.

Formally, the propagation of the hidden representations $\mathbf{H}^{(t)}$ among neighboring nodes for one time-step is formulated as:

$$\mathbf{H}^{(t)} = \left[\mathbf{h}_1^{(t)}, \ldots, \mathbf{h}_{|\mathcal{V}|}^{(t)}\right] \tag{5}$$

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \tag{6}$$

$$\mathbf{z}^{(t)} = \sigma\Big(\mathbf{W}_z x^{(t)} + \mathbf{U}_z \mathbf{H}^{(t-1)} + \mathbf{b}_z\Big) \tag{7}$$

$$\mathbf{r}^{(t)} = \sigma\Big(\mathbf{W}_r x^{(t)} + \mathbf{U}_r \mathbf{H}^{(t-1)} + \mathbf{b}_r\Big) \tag{8}$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\Big(\mathbf{W} x^{(t)} + \mathbf{U}\Big(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\Big) + \mathbf{b}\Big) \tag{9}$$

$$\mathbf{H}^{(t)} = \Big(1 - \mathbf{z}^{(t)}\Big) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \tag{10}$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}$ and $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}$ are trainable weight matrices, and $\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is the adjacency matrix, determining the edges in the graph $\mathcal{G}$.

Equation 6 corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for GNNs in Equations 1 and 4. Equations 7 to 10 correspond to the mathematical formulation of a GRU cell [43], with Equation 7 representing the GRU reset gate vector, Equation 8 the GRU update gate vector, and Equation 10 the GRU output.

In order to propagate the hidden representations throughout the complete graph, a fixed number of iterations of Equations 7 to 10 are performed. This extension has been shown to outperform standard GNN which require to run the recursion until a fixed point is found.

We also extended our neural network architecture with an edge attention mechanism similar to the one proposed in [45]. Thus, the neural network can give preference to some neighbors over other ones via a trained function. For each edge (v, u) in the graph, we define a weight parameter $\rho_{v,u}^{(t)}$ depending on the concatenation of $\mathbf{h}_v^{(t)}$ and $\mathbf{h}_u^{(t)}$:

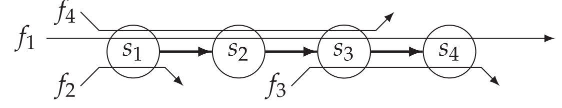


Fig. 2. Server Graph Model in NC.

$$\rho_{v,u}^{(t)} = \sigma\Big(\mathbf{W}_a \Big\{\mathbf{h}_v^{(t)}, \mathbf{h}_u^{(t)}\Big\} + \mathbf{b}_a\Big) \tag{11}$$

with trainable weights $\mathbf{W}_a$ and bias parameters $\mathbf{b}_a$. Equation 4 can then be rewritten as

$$\mathbf{h}_v^{(t)} = \sum_{u \in \mathrm{NBR}(v)} \rho_{v,u}^{(t-1)} f\Big(\mathbf{h}_u^{(t-1)}\Big). \tag{12}$$

B. Network Calculus

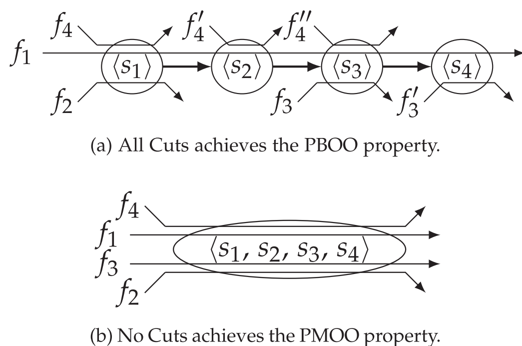
The NC model of a network is a directed graph called server graph $\mathcal{G}_{NC} = (\mathcal{V}_{NC}, \mathcal{E}_{NC}, \mathcal{F})$ with servers $s \in \mathcal{V}_{NC}$, edges $(v, u) \in \mathcal{E}_{NC}$ and data flows $f \in \mathcal{F}$. Servers represent the forwarding locations in a network, e.g., queues or packet schedulers. They guarantee a lower bound on data forwarding and thus an output quantity given an input quantity of data. Flows travel along directed edges, crossing servers and demanding their forwarding service. I.e., they define the input quantity of servers. NC models this with an upper bound on the flow's data arrivals valid in any duration of time. Figure 2 shows a server graph, a tandem.

Definition 1 (Tandem of Servers): A server graph $\mathcal{T} = (\mathcal{V}_{NC}, \mathcal{E}_{NC}, \mathcal{F})$ is called a tandem if the following properties hold: $|\mathcal{E}_{NC}| = |\mathcal{V}_{NC}| - 1$. For any server $s \in \mathcal{V}_{NC}$, let $in(s)$ be the amount of directed edges ending in server s and $out(s)$ be the amount of edges starting in s. On a tandem, it holds $\forall s \in \mathcal{V}_{NC} \mid \max(in(s)) = \min(1, \max(out(s))) = 1$.

In this paper, we put some additional assumptions on the NC model:

- There cannot be cyclic dependencies between flows. This is achieved by a restriction to feed-forward networks such as the tandem network of Figure 2. Any network can be converted to a feed-forward one [55].
- When multiple flows multiplex at a server, e.g., f_1, f_2 and f_4 at s_1 in Figure 2, we do not know the resulting order of their data. This is called arbitrary multiplexing.
- However, we assume that the order of data within a single flow will not change by multiplexing or forwarding.
- Flows are routed along point-to-point paths.
- The NC curves that bound data arrivals and forwarding capabilities are restricted to certain shapes: the minimum over multiple token buckets (like IntServ's TSPEC) and the maximum over multiple rate latencies, respectively. Details can be found in [16]–[18].

A network calculus analysis takes such a model as the input and computes a bound on a specific flow's end-to-end delay – as close to the realistic worst case as possible – with the least computational effort possible. The analyzed flow is called flow of interest (foi) and the set of tandem analyses has been



(a) All Cuts achieves the PBOO property.

(b) No Cuts achieves the PMOO property.

Fig. 3. Reduction rules in the NC analysis.

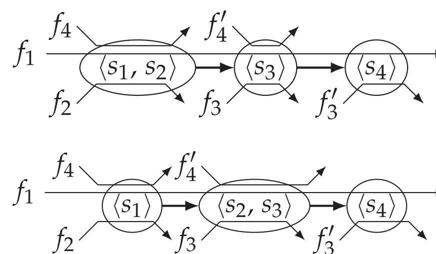


Fig. 4. Some additional reduction rules applied by TMA.

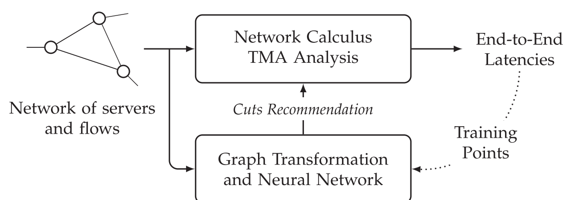


Fig. 5. Overview of the proposed approach.

steadily extended in order to improve tightness of the foi's delay bound. Two leaps have been taken by incorporating the Pay Bursts Only Once (PBOO) [14] and the Pay Multiplexing Only Once (PMOO) [15] properties into the NC analysis. Both mitigate previously added pessimism not found in a realistic system but required by the analysis. PBOO prevents the bound on the foi's worst-case burstiness to appear multiple times in the analysis as if it built up at every server – an unrealistic contention model. PMOO extends this to the burstiness of cross-traffic present on consecutive servers on the foi's path. In NC's interpretation as term rewriting [56], these two analyses are reduction rules. Their central means of transforming tandems is *cutting*.

Definition 2 (Cutting NC Tandems): Given a tandem $\mathcal{T} = (\mathcal{V}_{NC}, \mathcal{E}_{NC}, \mathcal{F})$ and a NC analysis $\mathcal{A}$, a cut marks edge $e \in \mathcal{E}_{NC}$ such that $\mathcal{A}$ will analyze $\mathcal{T}$ as a sequence of sub-tandems $\langle \mathcal{T}_l, \mathcal{T}_r \rangle$ where $\mathcal{T}_l$ holds all the model information to the left of e and $\mathcal{T}_r$ that to the right of e. A cutting (also called combination of cuts) is a set of cuts on $\mathcal{T}$.

Visually, the analyses implementing PBOO (Seperate Flow Analysis, SFA [14]) and PMOO (PMOO analysis, PMOOA [15]) proceed as depicted in Figure 3 a and 3 b:

All Cuts (Figure 3 a): SFA cuts every edge in the NC model along with the flows crossing it (except the foi f_1). The resulting sub-tandems are demarcated with $\langle \cdot \rangle$ and consist of single servers. For the cut flows, their arrivals at the subsequent server need to be bounded (we denoted the respective location with f_i'). Such a flow's bound consists of its initial – given burstiness – bound plus the worst-case increase due to having crossed the previous servers.

No Cuts (Figure 3 b): Without cuts, the entire tandem is analyzed at once. Mitigating the need for deriving bounds on flow arrivals in the network allows for achieving the PMOO property in addition to PBOO. For details on how to implement a no-cuts analysis, we refer the reader to [15].

The two reduction rules are part of the algebraic NC analysis branch. It was discovered that the algebraic analysis pays for its ability to apply the no-cuts reduction with the loss of server order information. As a consequence, neither of the reduction rules are generally resulting in a tighter delay bound than the other one. Therefore, the optimization-based NC analysis branch was proposed [16]. Later, a tight optimization analysis was developed [17], along with a heuristic that promises better scalability with increasing network size. In [19], it was shown that said heuristic may not scale well and that it is rivaled by algebraic NC in terms of delay bound tightness, too. The so-called Tandem Matching Analysis (TMA) we base our work on partially overcomes the computational effort challenges imposed by the exhaustive search over all combinations of the reduction rules above. See Figure 4 for two alternative rules to *all cuts* and *no cuts*. With DeepTMA, we make the approach scale even better by predicting the best tandem matching, i.e., combination of cuts, instead of exhaustively searching for it.

IV. Graph Neural Network for NC

We develop our DeepTMA heuristic in this section. It is based on the concept of GNN introduced in earlier. The goal of DeepTMA is to predict the best tandem decompositions, i.e., combinations of cuts, to use in TMA. For simplicity, we refer to NC server graphs as networks and to the graph model used in GNN as graphs.

The main intuition is to transform the NC server graph and flows into an undirected graph. This graph representation is then used as input for a neural network architecture able to process general graphs, which will then predict the tandem decomposition resulting in the best residual service curves. Our approach is illustrated in Figure 5. Since the delay bounds are still computed using the formal network calculus analysis, they inherit their provable correctness.

A. Application to TMA

In order to apply the concepts described in Section III-A to a network calculus analysis, we model NC's directed network as an undirected graph. Figure 6 illustrates this graph encoding on the network from Figure 2.

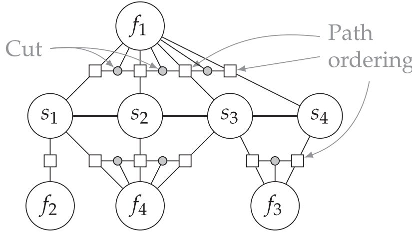


Fig. 6. Transformed network of Figure 2 to the graph model.

Each server is represented as a node in the graph, with edges corresponding to the network's links. Each flow is represented as a node in the graph, too. In order to encode the path taken by a flow in this graph, we use edges to connect the flow to the servers it traverses. Since those edges do not encode the order in which those servers are traversed, so-called *path ordering* nodes containing the hop count as feature are added to edges between the flow node and the traversed server nodes. This property is especially important in the TMA since the order, and hence position of cuts, has a large impact on dependency structures.

In order to represent these TMA cuts, each potential cut between pairs of servers on the path traversed by the flow is represented as a node. This cut node is connected via edges to the flow and to the pair of servers it is associated to.

In addition to a categorical encoding of the node type (i.e., server, flow, path ordering or cut), the input features of each node in the graph are as follows:

- For each server s, parameters of its rate-latency service curve $\beta_s(d) = \max\{0, rate_s \cdot d - latency_s\}$ are used: $[rate_s, latency_s]$
- For each flow f, parameters of its token-bucket arrival curve $\alpha_f(d) = \{rate_f \cdot d + burst_f\}_{\{d>0\}}$ (i.e., $\alpha_f(d) = 0$ for $d \leq 0$) are used: $[rate_f, burst_f]$
- For each path ordering p, the hop count is encoded as an integer: $PathOrder$
- Finally, neither cut nodes nor edges have input features

Equation 13 illustrates the matrix encoding of part of the graph from Figure 6.

$$\begin{bmatrix} \text{Server} & \text{Flow} & \text{Path Order} & \text{Cut} & \text{S. Rate} & \text{S. Latency} & \text{F. Rate} & \text{F. Burst} & \text{Hop count} \\ 1 & 0 & 0 & 0 & R_{s_i} & L_{s_i} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & r_{f_k} & b_{f_k} & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} \\ s_i \\ f_k \\ P_{f_1,s_1} \\ P_{f_1,s_2} \\ P_{f_1,s_3} \\ P_{f_1,s_4} \\ C^{f_k}_{s_i,s_j} \end{matrix} \tag{13}$$

Note, that the above restriction to (single) rate-latency curves for the service capabilities and (single) token-bucket curves for arrival constraints is not a restriction of our DeepTMA analysis. It is trivial to extend input features to the larger set of curve parameters required to model the curve shapes mentioned in Section III-B.

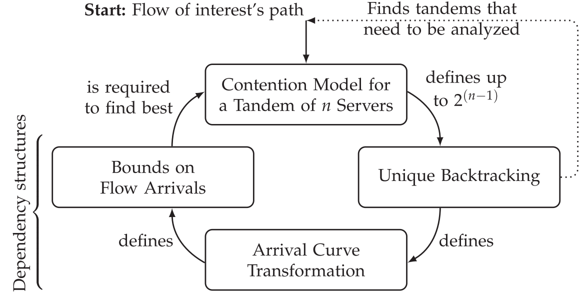


Fig. 7. Dependency cycle defining current NC analyses.

Based on this description of the server graph, the problem of choosing the best tandem decomposition to give to the NC analysis is formulated as a classification problem. Namely each cut node has to be classified in two classes: perform a cut between the pair of servers it is connected to or not: $[cut, \overline{cut}]$. The binary cross-entropy loss function is used during training for this classification problem. The other nodes of the graph are masked from the loss function.

The overall prediction to be fed back, i.e., the selection of one out of TMA's potential decompositions for a given foi's path, is defined by the set of all *cut* classifications for this path. The prediction of the best decomposition for a given tandem, starting with the foi's path, is done by iterating over all potential cuts and selecting the ones which have been classified as cutting points for said tandem.

B. Best Contention Models Across the Entire Analysis

Figures 3 and 4 in Section III-B already show the need for arrival bounding on the foi's path – see flow labels f_i' and f_i''. Moreover, our sample tandem assumes that bounds on flow arrivals are known when entering the tandem. This need not be the case in a feed-forward network where cross-flows traversed multiple servers before interfering with the flow of interest. Therefore cross-flow arrivals are required to be computed here, too. Bounding the arrivals of cross-traffic becomes a resource intensive, recursive procedure [5], [57]. It starts with the foi and it only terminates in feed-forward networks when all cross-flows are backtracked to their sources. The procedure is visualized in Figure 7.

Applying the exhaustive TMA in every recursion level (i.e., every cycle in Figure 7) yields large computational demands. Given a tandem of length n servers, TMA tests all $2^{(n-1)}$ combinations of cuts. Visually, TMA unwinds all loops and branches (see dashed line) that can be taken in the cycle. On the other hand, the minimum-cost NC analysis can be obtained by unwinding only the bare minimum of loops: do not take optional branches by choosing a single contention model like PBOO (SFA) or PMOO do. With DeepTMA, we create an alternative heuristic that is not deciding on the cut

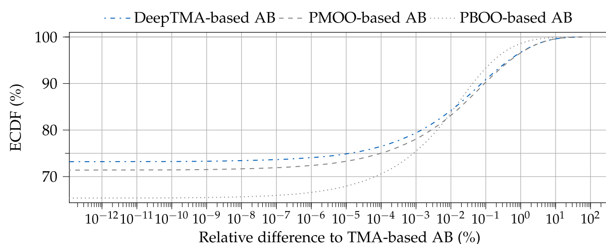


Fig. 8. Comparison of arrival bounding (AB) methods on the dataset constructed in Section V-A.

combination in a static way. Instead, our GNN uses a range of input features to predict the best cut combination.

The original DeepTMA of [11] was designed to predict the best cuts for a tandem of servers, subject to minimizing the analyzed flow's delay bound. However, the delay bound is only computed in the very first iteration of Figure 7's loop (after "**Start:** Flow of interest's path"). As seen above, any subsequent iteration will be part of the arrival bounding of cross flows. In this article, we evolve DeepTMA to also provide delay-bound-minimizing cut combinations for the bounding cross-traffic arrivals. Our delay-bound-based approach has the advantage of not instantiating, training and integrating a second GNN for the purpose of arrival bounding. Figure 8 shows that our approach is indeed superior in the vast amount of cases and we will apply this DeepTMA-based arrival bounding in all our evaluations.

V. Implementation and Dataset Generation

We implemented DeepTMA and the graph neural network architecture using PyTorch [58]. The recursion from Equation 1 was dynamically unrolled for a fixed number of iteration according to the diameters of the analyzed graphs. Table I illustrates the size of the different layers used here.

We analyzed each network with the NetworkCalculus.org Deterministic Network Calculator (NCorg DNC),[2] version 2.6.0 and perform the exhaustive TMA analysis to generate the best cuts combinations. A tandem decomposition is always executed for a flow of interest. But instead of the residual service curves, we use the delay bounds for the foi as caused by all decompositions in order to rank them. This is because the former potentially faces problems in the case of lost service curve strictness.

A. Dataset Generation

In order to train our neural network architecture, we follow a traditional supervised learning approach. We randomly generated a set of random topologies according to three different random topology generators:

(1) tandems or daisy-chains like in Figure 2,
(2) trees and
(3) random server graphs following the Erdős Rényi model [59], then made feed-forward with the Turn Prohibition algorithm [55].

[2] Formerly known as DiscoDNC [21] see networkcalculus.org/dnc

TABLE I
Size of the Layers Used in the GGNN. Indexes Represent Respectively the Weights (w) and Biases (b) Matrices

Layer	NN Type	Size
init	FFNN	$(21,160)_w + (160)_b$
Memory unit	GRU cell	$(320,320)_w + (320,160)_w + (480)_b$
Edge attention	FFNN	$(320,1)_w + (2)_b$
out hidden layers	FFNN	$2 \times \{(160,160)_w + (160)_b\}$
out final	FFNN	$(160,2)_w + (2)_b$
	Total:	209 764 parameters

TABLE II
Statistics About the Generated Dataset

Parameter	Min	Max	Mean	Median
# of servers	2	41	14.6	12
# of flows	3	203	101.2	100
# of tandem combinations	2	197 196	1508.5	384
# of nodes in analyzed graph	10	2093	545.2	504
# of tandem combination per flow	2	65 536	19.4	4
# of flows per server	1	173	18.1	10

For each created server, a rate latency service curve was generated with rate and latency parameters taken from a uniform distribution. A random number of flows with random source and sink servers was added. For each flow, a token bucket arrival curve was generated with burst and rate parameters taken from a uniform distribution. All curve parameters were normalized to the $(0,1]$ interval.

In total, 172374 different networks were generated, with a total of more than 13 million flows, and close to 260 million tandem decompositions. Half of the networks were used for training the neural network, while the other half was used for the evaluation presented later in Section VII. Table II summarizes different statistics about the generated dataset. The dataset is available online[3] to reproduce our learning results.

VI. Other TMA Heuristics

To benchmark DeepTMA, we present three additional heuristics for the choice of TMA's tandem decompositions. Compared to the GNN-based proposal, those heuristics are based on simpler algorithms.

A. RND: Random Choice of Tandem Decomposition

The simplest heuristic is to randomly select multiple alternative tandem decompositions, where each decomposition has the same probability of being chosen. Given any n-server tandem, starting with the foi's path as shown in Figure 7, RND only selects $n' \ll 2^{(n-1)}$ decompositions. I.e., the RND heuristic randomly samples a small part of TMA's search space per tandem in the analysis. The remainder of the analysis follows the standard NC proceeding.

[3] https://github.com/fabgeyer/dataset-deeptma-extension

Algorithm 1: Generation of n tandem decompositions for a flow traversing $L+1$ servers.

if $n \leq 2^L$ **then**
return all combinations of cuts
else
  for all $i := 1$ **to** n **do**
    $v \leftarrow [c_1, \ldots, c_L] \sim \mathcal{U}(0,1)^L$
    $\text{cuts}_i \leftarrow \mathbb{I}(v \leq [\Pr(cut^{GNN}_{foi,1}), \ldots, \Pr(cut^{GNN}_{foi,L})])$
      ($\mathbb{I}$ is the indicator function)
  endfor
  return $\{\text{cuts}_1, \ldots, \text{cuts}_n\}$
end if

B. Simplified Machine Learning Heuristics

While DeepTMA is based on an approach which uses the complete information about the server graph, we propose here a simpler machine-learning approach which uses a simplified view of the server graph and its features. As for DeepTMA, this heuristic uses machine learning algorithms in order to classify each cut in the same two classes, namely decide to perform a cut between a pair of servers or not.

This simplified approach only uses a local view of the network, i.e. parameters of the pair of servers between which the cut is located, named here *source* and *sink*. For each cut in the network, we define a feature vector comprised of the following values:

- *FlowArrival{Rate,Burst}*: the parameters of the token bucket arrival curve of the foi;
- *FlowPathLen:* the path length of the foi;
- *CutOrder:* the index of the cut in the path of the foi;
- *{Source,Sink}Service{Rate,Latency}*: the parameters of the rate-latency service curves of the pair of servers;
- *{Source,Sink}SumArrival{Rate,Burst}*: the sum of the parameters of the arrival curves of the flows traversing each server of the pair;
- *SourceNFlows* and *SinkNFlows:* the number of flows at each server of the pair.

While this simplified view of the server graph performs worse than the one used DeepTMA – as show later in Section VII – our main motivations for this simplified approach are simplicity and explainability of the model. Feature importance [60], [61] is more easily performed on such simplified feature vector than on the GNN model, as illustrated later in Section VIII.

Using those feature vectors, we propose here two heuristics, one based on feed-forward neural network, and one based on random forests. Since the output of both heuristics is a probability of making a cut, multiple tandem decompositions can be generated using the approach presented later in Section VI-C and Algorithm 1.

1) FFNN: Feed-Forward Neural Network Heuristic: This heuristic uses a standard multi-layer feed-forward neural network to classify the cuts using the simplified feature vector presented earlier. The size and number of hidden layers of the FFNN is detailed in Table III. We use standard training

TABLE III
SIZE OF THE LAYERS USED IN THE FFNN. INDEXES REPRESENT RESPECTIVELY THE WEIGHTS (w) AND BIASES (b) MATRICES

Layer	Size
input	$(10,64)_w + (64)_b$
hidden layers	$2 \times \{(64,64)_w + (64)_b\}$
out final	$(64,2)_w + (2)_b$
Total:	9154 parameters

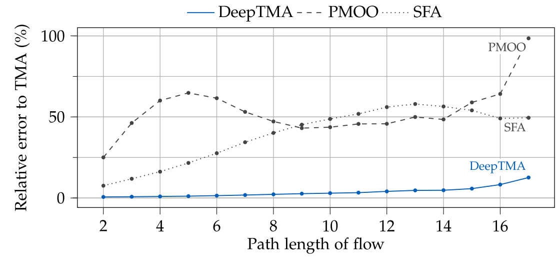


Fig. 9. Relative error of DeepTMA and existing NC heuristics.

method based on gradient descent to train the neural network. As for DeepTMA, our implementation is based on PyTorch [58].

2) RFC: Random Forest Classifier Heuristic: This heuristic uses random forests [60] to classify the cuts using the simplified feature vector presented earlier. Our implementation of this heuristic is based on scikit-learn [62].

C. Generating Multiple Decompositions

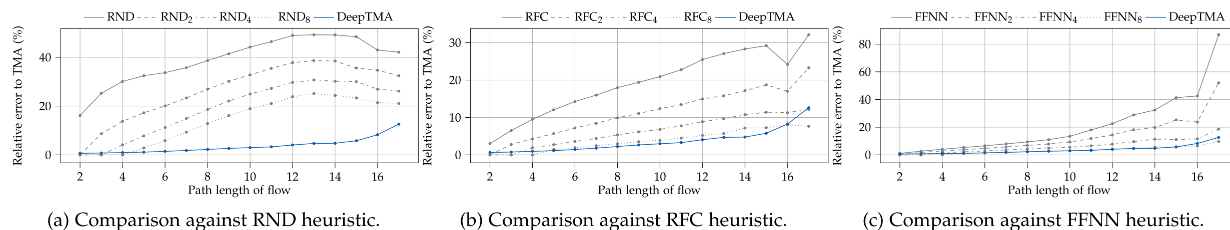
Given a foi and a cut, the output of the machine learning-based heuristics presented earlier is a probability of cutting. This probability is generated by the neural networks using the softmax function after the last layer. In case a single tandem decomposition has to be generated, the decision of cutting or not is made using a threshold of 50%.

The cut probabilities may also be used in order to generate multiple tandem decompositions as illustrated in Algorithm 1. In case the number of tandem decompositions is lower than the number of requested decompositions, we simply return all combinations of cuts. Otherwise, we sample the distribution of cuts in order to generate the decompositions. In Section VII, we label those extended heuristics using n as subscript, with n the number of decompositions.

VII. NUMERICAL BENCHMARKS

We evaluate in this section DeepTMA against classical NC analyses, TMA, and the heuristics presented above. Via a numerical evaluation, we illustrate the tightness and execution time, and highlight the usability for practical use-cases.

Unless specified otherwise, all the evaluations presented in this section were performed on the dataset described in Section V-A. In order to perform the evaluation, the dataset was split in two parts: one part was used for training the machine learning-based heuristics, while the second part was used to perform the numerical evaluations presented in this section. Additionally, DeepTMA



(a) Comparison against RND heuristic. (b) Comparison against RFC heuristic. (c) Comparison against FFNN heuristic.

Fig. 10. Relative error of DeepTMA and the heuristics presented in Section VI.

TABLE IV
STATISTICS ABOUT THE SET OF NETWORKS FROM [19]

Parameter	Min	Max	Mean	Median
# of servers	38	3626	863.0	693
# of flows	152	14 504	3452.0	2772
# of tandem combinations	2418	121 860	24 777.6	18 869
# of nodes in analyzed graph	1358	113 162	25 137.7	19 518
# of tandem combination per flow	2	512	7.3	8
# of flows per server	1	467	16.4	12

was also evaluated on the set of network from [19] in Section VII-B, and on the set of networks from [11] in Section VII-D.

A. Relative Error

We investigate in this section the resulting loss of tightness in case a non-optimal tandem decomposition was selected by a given heuristic. In order to quantitatively evaluate this loss of tightness compared to TMA, we use the relative error, defined as:

$$RelErr_{\text{foi}} = (delay_{\text{foi}}^{\text{heuristic}} - delay_{\text{foi}}^{\text{TMA}})/delay_{\text{foi}}^{\text{TMA}} \quad (14)$$

Classical NC Analyses: Figure 9 illustrates the relative error of DeepTMA against classical NC analyses. DeepTMA-derived delay bounds are tightest among these heuristics, deviating from TMA by no more than 12% in our experiments in the worst-case.

DeepTMA efficacy beating SFA and PMOO in the cost/ tightness-tradeoff is necessary, yet, by no means sufficient to conclude that our deep-learning assisted analysis framework is the best alternative to create heuristics. SFA and PMOO were created a decade before TMA, i.e., they never benefited from advances that resulted in TMA. Therefore, we base our statement on numerical benchmarks against newly contributed ML-based heuristics for TMA.

New Heuristics: Figure 10 compares DeepTMA against the other heuristics introduced in Section VI. Only FFNN_8 and RFC_8 are able to achieve a relative error similarly small as DeepTMA on the larger networks, yet, at a much larger computational cost since 8 different tandem combinations and their entire dependency structures have to be evaluated every time.

B. Scalability and Robustness on Larger Networks

Additionally to the dataset which was presented in Section V-A, we also evaluate our approach on the set of networks used in [19]. No additional training of the GNN is performed on this additional dataset. Table IV summarizes

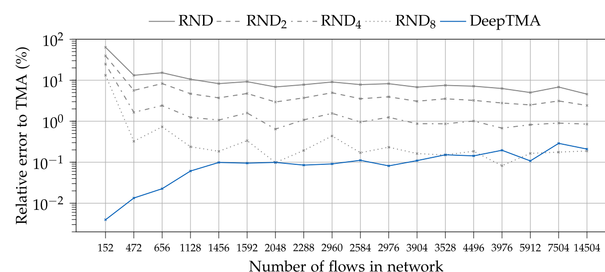


Fig. 11. Relavitve error of DeepTMA on the dataset from [19] with much larger networks.

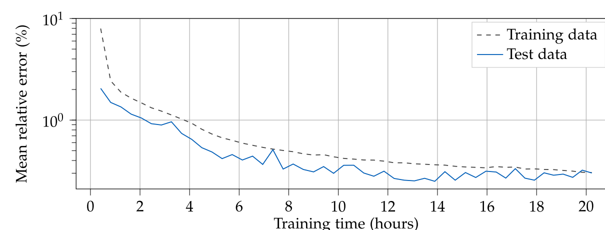


Fig. 12. Evolution of the relative error during training.

different statistics about this additional dataset. Compared to dataset used for training, this additional set of networks is up to two order of magnitude larger in term of number of servers and flows per network. We evaluate here if our approach is able to scale to such larger networks in terms of tightness.

Figure 11 illustrates the relative error of DeepTMA compared to a random heuristic which selects the tandem decompositions randomly. DeepTMA achieves relative errors that are two orders of magnitudes smaller than the random heuristics, resulting in better end-to-end delay bound accuracy w.r.t. the exhaustive TMA. Although DeepTMA was not trained on such large networks, the relative error still stays below .3% even on the larger networks. Those results highlight that DeepTMA is indeed able to scale to networks much larger than to those it was initially trained for.

Additional results regarding robustness of DeepTMA with respect to scalability on larger networks can also be found in [24].

C. Training Time

We illustrate in Figure 12 the evolution of the relative error during the training phase of the GNN. As noted in Section V-A, this training was done 86187 topologies.

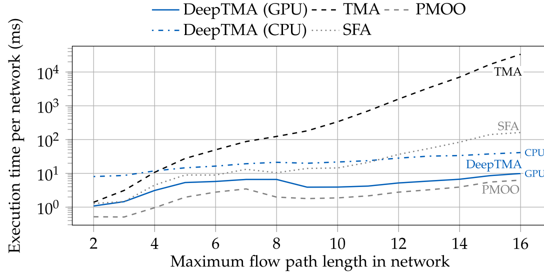


Fig. 13. Comparing DeepTMA to existing NC heuristics on the dataset from [11].

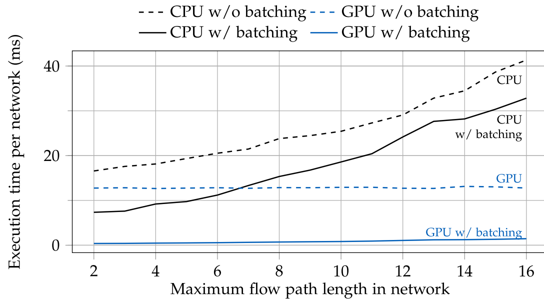


Fig. 14. Execution time of the cut recommendation part of DeepTMA, executed on CPU or GPU, without batching or batch sizes of 64 networks on the dataset from [11].

Training duration was measured while training was done using a Nvidia GTX 1080 Ti GPU.

D. Execution Times

In order to understand the practical applicability of our heuristic, we evaluate in this section its execution time in different settings. We define and measure the execution time per network as the total time taken to process N networks and all its flows divided by N, without including the startup time or the time taken for initializing the network data structures.

Classical NC Analyses: Figure 13 shows benchmarking results of DeepTMA against the classical analysis in NC. We compare against TMA and the established SFA [14] and PMOO [15] heuristics of NC. These are fast as they greedily decide on a single contention model, ignoring arrival and service curves. DeepTMA from our framework is minimally slower than PMOO but faster than SFA and TMA. This implies two things: first, the overhead of querying for predictions is not necessarily large and secondly, the contention model tends be closer to PMOO than to SFA, consisting of tandems of multiple servers.

TMA: Since DeepTMA can be executed on either CPU or GPU, we first compare both platforms and their affinity at parallelization in Figure 14. A Nvidia GTX 1080 Ti was used for the measurements on GPU, and an Intel Xeon E3-1270 v6 (at 3.80 GHz) for the ones on CPU. We first notice that the execution time grows close to linearly with the size of the network, both on CPU and GPU, which is explained by the iterations of message passing illustrated in Equation 6 according to the diameter of the studied graph. Execution on GPU results in faster computation compared to CPU for networks larger than two hops, mainly due to the better ability of GPUs of parallelizing the numerical operations used in neural networks.

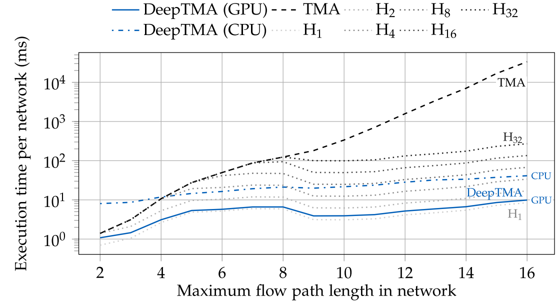


Fig. 15. Execution times per topology for TMA, DeepTMA and H_n heuristics on the dataset from [11].

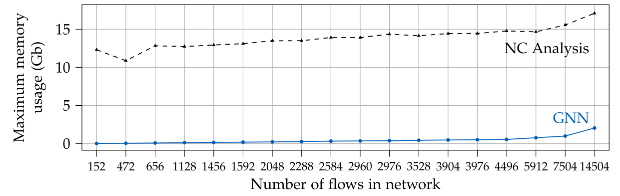


Fig. 16. Memory usage of the GNN and the TMA NC analysis on the dataset from [11].

Since both platforms offer multiple cores for parallel execution of multiple processes, we investigate the effect of batching, namely analyzing multiple networks in parallel. Parallelization of the mathematical operations described in Section IV is automatically performed by PyTorch. We present in Figure 14 the execution time without any batching – namely only one network is processed at once – and with batching, where the heuristic processes 64 networks at once. On both platforms, batching results in a reduction of processing time, which is relevant in use-cases where multiple network configuration have to be processed.

In addition, we measured the execution time of TMA using the NCorg DNC [21]. The same CPU was used for running NCorg DNC, with Oracle's HotSpot JVM version 1.8.

Whereas Figure 14 provides insight on the computational cost of DeepTMA, Figure 15 compares it to a generalized version of the heuristics presented in Section VI. Since the selection of tandem decompositions is a fast operation in all three pure NC heuristics, in particular compared to the other required operations, we only illustrate the execution time of a generic heuristic H_n selecting n decompositions per tandem. As all analyses ultimately use the NCorg DNC for the derivation of bounds, comparing the average execution times of H_n and DeepTMA (with batching), we can also judge the increase of computational effort due to our deep learning-based predictions. As expected, TMA execution times grow exponentially and H_n heuristics' execution times coincide with TMA as

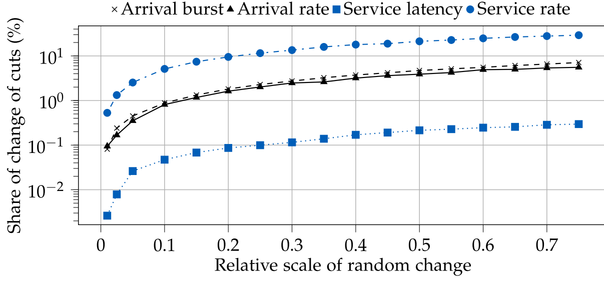


Fig. 17. Sensitivity analysis of the TMA cut choices.

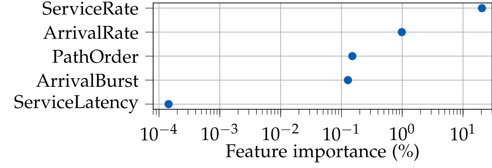


Fig. 18. Feature importance according to feature permutation method for DeepTMA.

long as their n-value causes an exhaustive search, too. An entirely CPU-bound DeepTMA analysis is slowest in very small networks where the exhaustive enumeration of TMA is easily possible to execute. Starting at a maximum flow path length of 4, it mostly performs between H_4 and H_8. Yet, we saw in Section VII-A that RND_i $i \in 4, 8$ is outperformed by DeepTMA. DeepTMA leveraging GPU technology for predictions only adds very small execution times to H_1 while achieving vastly better bounds. Compared to TMA, we can observe a measured differences in execution time growing up to four orders of magnitude.

E. Memory Footprint

We evaluate in Figure 16 the memory footprint of the classical TMA against the GNN heuristic. Compared to the classical NC analysis, the GNN requires almost an order of magnitude less memory, even on the larger networks with up to 14504 flows. In summary, those results and the results from Section VII-D illustrate that DeepTMA requires only a fraction of the computing and memory resources of the classical analysis, with only a minor drop in tightness.

VIII. Insights Into Learning Approaches

In order to better understand the importance of the input features used in DeepTMA and the two other machine learning-based heuristics proposed in this article, we perform in this section a sensitivity analysis of TMA and assess DeepTMA's features' importance.

A. Sensitivity Analysis

We perform here a sensitivity analysis of TMA's cut choices in order to better understand which parameters influence the choice of best cuts. To numerically evaluate the sensitivity, we randomly modify the service and arrival curve parameters $p_{original}$ of our evaluation networks with a relative scale s according to the following uniform distribution:

$$p_{new} \sim \mathcal{U}\big(p_{original}(1-s), p_{original}(1+s)\big) \quad (15)$$

We then compare the share of flows where best cuts have changed due to the random change of curves parameters. Results are presented in Figure 17.

The service rate parameter influences the most the choice of cuts, where even minimal changes result in different choices. Arrival curve parameters also impact also TMA's analysis, but with less magnitude than the service rate. Finally, the service latency has almost no influence on the choice of cuts, where even large changes of its value result in less than 1% of changed choices.

These observations can be explained by the fundamental impact a cut in TMA has on the composition of sub-tandem residual service that is composed to an end-to-end guaranteed service for the foi. On any tandem, forwarding is lower bounded by the minimum residual rate over all servers. Separating a subset of faster or slower servers by cutting can therefore easily enable to make use of larger residual service on separated sub-tandems, in particular when bounding cross-flow arrivals. The service latency, in contrast, is an additive factor in the residual forwarding service computation, making it considerably less impactful. Cutting a link traversed by a cross-flow means computing an arrival curve for said flow at this location, an output bound after having traversed the previous sub-tandem. These output bounds consist of the original burstiness and a burstiness increase due to queueing. Thus, a significant change in the burstiness increase is required to change the choice of cut set. This is less likely to be achieved by solely modifying original arrival curves' rate or burstiness parameters.

B. Feature Importance

We use the permutation-based importance measure [60], [61] in order to assess each feature's importance of DeepTMA. For each input feature presented in Sections IV-A and VI-B, we randomize it by randomly permuting its values in the training set, and assess the impact it has on the relative error of the predictions. We define the importance metric as:

$$Importance(Feature) = \frac{1}{|\mathcal{F}|}\sum_{f_i \in \mathcal{F}}\Big(RelErr_{f_i}^{Feature} - RelErr_{f_i}^{Baseline}\Big) \quad (16)$$

with the baseline corresponding to the method without any feature permutation. With this evaluation, we assess how much the GNN model relies on a given feature of interest for making its prediction.

DeepTMA's features importance are presented in Figure 18. The service rate of the servers in the network have the largest impact on the final decision of cutting. The remaining features appear to have less importance on the cut prediction, with

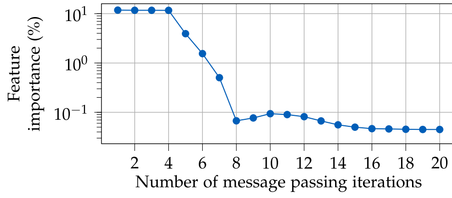


Fig. 19. Importance of message passing iterations.

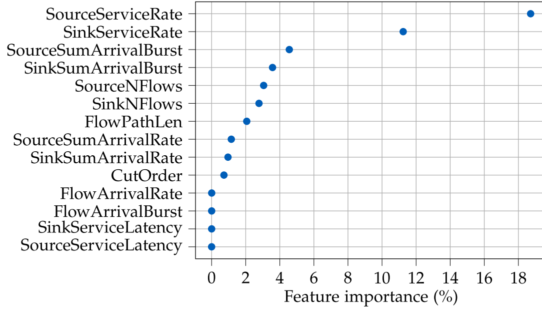


Fig. 20. Feature importance according to feature permutation method for the FFNN heuristic.

almost no importance for the service latency. These results confirm the findings presented in Figure 17, showing that the GNN matches TMA's sensitivity. Figure 18 also highlights that the arrival rate is of considerably larger importance than the arrival burstiness whereas the sensitivity analysis showed slightly more impact of a changing burstiness. The importance of the arrival rates is natural to NC with arbitrary multiplexing: the derived worst-case assumption is that the flow of interest is served with the residual forwarding capabilities and the according service curve's latency increases fast with increasing utilization. Thus, the (cross-traffic) arrival rates must be an important feature.

Interesting from the NC perspective is the observation that the order of servers (PathOrder) has a percental importance two orders of magnitude lower than the service rate. In combination, these two features constitute the very reason for TMA (and optimization-based analyses, see [16]) to outperform the previous NC analyses.

We also assess the importance of other flows and other servers on a cut. We perform this by assessing the number of iterations of message passing (i.e. Equation 1) and the impact it has on the relative error. As for feature importance, we compare the results according to Equation 16. Results are presented in Figure 19. The first 4 loop iterations appear to have the largest impact on the cut decision, meaning that the cut decision is mainly based on information from servers close to the cut. We notice that the importance drops sharply after 5 iterations, and converges after 15 iterations. This indicates that servers and flows farther away from the cut decision are less relevant to the cut decision – an insight to potential further improvement of DeepTMA's tradeoff between computational effort and relative error.

We evaluate also the features importance of the FFNN heuristic in Figure 20. Overall, those results confirm the ones

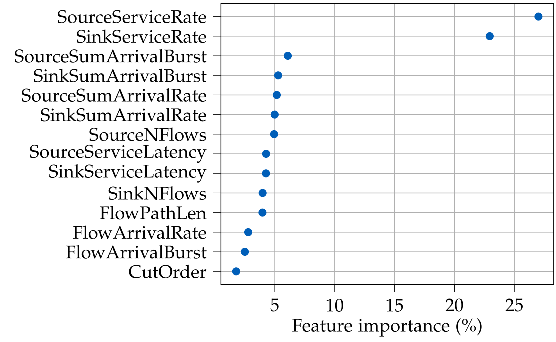


Fig. 21. Feature importance according to feature permutation method for the random forests classifier.

presented for DeepTMA, namely that the service rate of the servers have the largest impact on the cutting decision. Interestingly, the impact of the other flows is more related to the aggregated arrival burst (*SourceSumArrivalBurst* and *SinkSumArrivalBurst*) than the aggregated arrival rate compared to DeepTMA. Finally, as for DeepTMA, the position of the cut in the path of the flow is not relevant.

Finally, we also evaluate the features importance of the RFC heuristic in Figure 21. The results for the RFC heuristic are in line with the ones from DeepTMA and exhibit as similar ranking than for the FFNN. Overall, Figures 18, 20, and 21 confirm existing results of the NC analysis presented in Section VIII-A, namely its sensitivity to service rate and the importance of other flows.

IX. Conclusion

We contribute a new framework that combines network calculus and deep learning. The first heuristic created with our framework is the DeepTMA, deep learning-assisted TMA, a fast network analysis for deterministic end-to-end delay bounds. It solves the main bottleneck of the existing TMA, namely its exponential execution time growth with network size, by using predictions for effectively selecting the contention models in the network calculus analysis.

Via a numerical evaluation, we show that our heuristic is accurate and produces end-to-end delay bounds which are almost as tight as TMA, with an execution time several orders of magnitude smaller than TMA and a memory footprint an order of magnitude smaller. DeepTMA is as fast as or faster than previously widespread methods – namely SFA and PMOO – even when analyzing larger networks, but with a gain in tightness exceeding 50% in some cases. Numerical evaluations on large networks with up to 14000 flows also illustrate that our approach is able to scale despite having being trained on much smaller networks.

Our work is based on a transformation of the network of servers and flows crossing them into a graph which is analyzed using Graph Neural Networks. Our method outperforms simpler ML-based methods, justifying the use of a more complex machine learning method. Finally some insights into the learning is also given via an evaluation of feature importance, confirming existing results of NC.

REFERENCES

[1] F. Geyer and G. Carle, "Network engineering for real-time networks: Comparison of automotive and aeronautic industries approaches," *IEEE Commun. Mag.*, vol. 54, no. 2, pp. 106–112, Feb. 2016.

[2] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *Proc. IEEE Global Commun. Conf.*, 2013, pp. 1397–1402.

[3] X. Jin, N. Guan, J. Wang, and P. Zeng, "Analyzing multimode wireless sensor networks using the network calculus," *J. Sensors*, vol. 2015, pp. 1–12, Art. no. 851608.

[4] J. W. Guck, A. Van Bemten, and W. Kellerer, "DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 4, pp. 1003–1017, Dec. 2017.

[5] S. Bondorf and J. B. Schmitt, "Boosting sensor network calculus by thoroughly bounding cross-traffic," in *Proc. IEEE INFOCOM*, 2015, pp. 235–243.

[6] S. Vastag, "Modeling quantitative requirements in SLAs with network calculus," in *Proc. 5th Int. ICST Conf. Perform. Eval. Methodologies and Tools ValueTools*, 2011, pp. 391–398.

[7] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.

[8] E. J. Rosensweig and J. Kurose, "A network calculus for cache networks," in *Proc. IEEE INFOCOM*, 2013, pp. 85–89.

[9] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 435–448.

[10] A. Charny and J.-Y. Le Boudec, "Delay bounds in a network with aggregate scheduling," in *Proc. Int. Workshop Qual. Future Internet Services*, 2000, pp. 1–13.

[11] F. Geyer and S. Bondorf, "DeepTMA: Predicting effective contention models for network calculus using graph neural networks," in *Proc. of INFOCOM*, 2019, pp. 1009–1017.

[12] R. L. Cruz, "A calculus for network delay, part I: Network elements in isolation," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 114–131, Jan. 1991.

[13] R. L. Cruz, "A calculus for network delay, part II: Network analysis," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 132–141, Jan. 1991.

[14] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Germany: Springer-Verlag, 2001.

[15] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *Proc. Conf. Meas. Modell. Eval. Comput. Commun. Syst.*, 2008, pp. 1–15.

[16] J. B. Schmitt, F. A. Zdarsky, and M. Fidler, "Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch.," in *Proc. IEEE 27th Conf. Comput. Commun.*, 2008, pp. 1669–1677.

[17] A. Bouillard, L. Jouhet, and É. Thierry, "Tight performance bounds in the worst-case analysis of feed-forward networks," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[18] A. Bouillard and G. Stea, "Exact worst-case delay in FIFO-multiplexing feed-forward networks," *IEEE/ACM Trans. Net.*, vol. 23, no. 5, pp. 1387–1400, Oct. 2015.

[19] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proc. ACM Meas. Anal. Comput. Syst. (POMACS)*, vol. 1, no. 1, 2017, pp. 1–34.

[20] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2005, pp. 729–734.

[21] S. Bondorf and J. B. Schmitt, "The DiscoDNC v2 – A comprehensive tool for deterministic network calculus," in *Proc. 8th Int. Conf. Perform. Eval. Methodol. Tools*, 2014, pp. 44–49.

[22] F. Geyer, "Performance evaluation of network topologies using graph-based deep learning," in *Proc. 11th EAI Int. Conf. Perform. Eval. Methodologies Tools*, 2017, pp. 20–27.

[23] A. Scheffler, M. Fögen, and S. Bondorf, "The deterministic network calculus analysis: Reliability insights and performance improvements," in *Proc. IEEE 23rd Int. Workshop Comput. Aided Modeling Des. Commun. Links Netw.*, 2018, pp. 1–6.

[24] F. Geyer and S. Bondorf, "On the robustness of deep learning-predicted contention models for network calculus," in *Proc. IEEE Symp. Comput. Commun.*, 2020, pp. 1–7.

[25] F. Wang, Z. Cao, L. Tan, and H. Zong, "Survey on learning-based formal methods: Taxonomy, applications and possible future directions," *IEEE Access*, vol. 8, pp. 108561–108578, 2020.

[26] H. Al-Zubaidy, J. Liebeherr, and A. Burchard, "A (min, $\times$) network calculus for multi-hop fading channels," in *Proc. IEEE INFOCOM*, 2013, pp. 1833–1841.

[27] J. Liebeherr, "Duality of the max-plus and min-plus network calculus," *Found. Trends. Netw.*, vol. 11, no. 3-4, pp. 139–282, 2017.

[28] M. Boyer and P. Roux, "Embedding network calculus and event stream theory in a common model," in *Proc. IEEE 21st Int. Conf. Emerg. Technologies Factory Automat.*, 2016, pp. 1–8.

[29] K. Lampka, S. Perathoner, and L. Thiele, "Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems," in *Proc. 9th ACM IEEE Int. Conf. Embedded Softw.*, 2009, pp. 107–116.

[30] C.-S. Chang, *Performance Guarantees in Communication Networks*. Berlin, Germany: Springer, 2000.

[31] F. Ciucu, A. Burchard, and J. Liebeherr, "A network service curve approach for the stochastic analysis of networks," in *Proc. ACM SIGMETRICS*, 2005, pp. 279–290.

[32] F. Ciucu, F. Poloczek, and J. B. Schmitt, "Sharp per-flow delay bounds for bursty arrivals: The case of FIFO, SP, and EDF scheduling," in *Proc. IEEE INFOCOM*, 2014, pp. 1896–1904.

[33] M. A. Beck, S. A. Henningsen, S. B. Birnbach, and J. B. Schmitt, "Towards a statistical network calculus – dealing with uncertainty in arrivals," in *Proc. IEEE INFOCOM*, 2014, pp. 2382–2390.

[34] F. Dong, K. Wu, and V. Srinivasan, "Copula analysis for statistical network calculus," in *Proc. IEEE INFOCOM*, 2015, pp. 1535–1543.

[35] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. Int. Symp. Circuits Syst.*, 2000, pp. 101–104.

[36] N. Guan and W. Yi, "Finitary real-time calculus: Efficient performance analysis of distributed embedded systems," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 330–339.

[37] K. Lampka, S. Bondorf, and J. B. Schmitt, "Achieving efficiency without sacrificing model accuracy: Network calculus on compact domains," in *Proc. IEEE 24th Int. Symp. Modeling, Anal. Simul. Comput. Telecommun. Syst.*, 2016, pp. 313–318.

[38] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan, and W. Yi, "Generalized finitary real-time calculus," in *Proc. IEEE INFOCOM*, 2017, pp. 1–9.

[39] S. K. Khangura, M. Fidler, and B. Rosenhahn, "Neural networks for measurement-based bandwidth estimation," in *Proc. IFIP Netw. Conf.*, 2018, pp. 1–9.

[40] Q. Xu, J. Wang, and K. Wu, "Learning-based dynamic resource provisioning for network slicing with ensured end-to-end performance bound," *IEEE Trans. Netw. Sci. Eng.*, vol. 7, no. 1, pp. 28–41, 2020.

[41] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[42] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–20.

[43] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1724–1734.

[44] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. Neural Inf. Process. Syst.*, 2017, pp. 1263–1272.

[45] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–12.

[46] P. W. Battaglia *et al.*, "Relational inductive biases, deep learning, and graph networks," 2018, *arxiv:1806.01261*.

[47] D. K. Duvenaud *et al.*, "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. Neural Inf. Process. Syst.*, 2015, pp. 2224–2232.

[48] M. Prates, P. H. C. Avelar, H. Lemos, L. C. Lamb, and M. Y. Vardi, "Learning to solve NP-complete problems: A graph neural network for decision TSP," in *Proc. Conf. AI*, 2019, pp. 4731–4738.

[49] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–11.

[50] K. Rusek and P. Cholda, "Message-passing neural networks learn Little's law," *IEEE Commun. Lett.*, vol. 23, no. 2, pp. 274–277, 2019.

[51] F. Geyer, "DeepComNet: Performance evaluation of network topologies using graph-based deep learning," *Elsevier Perform. Eval.*, vol. 130, pp. 1–16, 2018.

[52] K. Rusek, J. Surez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, “RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN,” *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2260–2270, 2020.

[53] T. Suzuki, Y. Yasuda, R. Nakamura, and H. Ohsaki, “On estimating communication delays using graph convolutional networks with semi-supervised learning,” in *Proc. IEEE Int. Conf. Inf. Netw.*, 2020, pp. 481–486.

[54] F. Geyer and G. Carle, “The case for a network calculus heuristic: Using insights from data for tighter bounds,” in *Proc. 30th Int. Teletraffic Congr.*, 2018, pp. 43–48.

[55] D. Starobinski, M. Karpovsky, and L. A. Zakrevski, “Application of network calculus to general topologies using turn-prohibition,” *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 411–421, Jun. 2003.

[56] M. Boyer, “NC-maude: A rewriting tool to play with network calculus,” in *Proc. Int. Symp. Leveraging Appl. Formal Methods, Verification Validation*, 2010, pp 137–151.

[57] S. Bondorf and J. B. Schmitt, “Calculating accurate end-to-end delay bounds – You better know your cross-traffic,” in *Proc. 9th EAI Int. Conf. Perform. Eval. Methodol. Tools*, 2015, pp. 17–24.

[58] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. Adv. Conf. Neural Inf. Process. Syst. 32*, 2019, pp. 8024–8035.

[59] P. Erdős and A. Rényi, “On random graphs. i,” *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[60] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[61] A. Fisher, C. Rudin, and F. Dominici, “All models are wrong, but many are useful: Learning a variable’s importance by studying an entire class of prediction models simultaneously,” *J. Mach. Learn. Res.*, vol. 20, no. 177, pp. 1–81, 2019.

[62] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2826–2830, 2011.

Fabien Geyer received the master’s degree of engineering in telecommunications from Telecom Bretagne, Brest, France, in 2011 and the Ph.D. degree in computer science from Technische Universität München, Munich, Germany, in 2015. He is currently with Airbus Central Research & Technologies and the Technical University of Munich working on methods for network analytics, network performances, and architectures. His research interests include novel methods for data-driven networking, formal methods for performance evaluation, and modeling of networks.

Steffen Bondorf received the Dr.-Ing. in computer science from TU Kaiserslautern, Germany, in 2016. He is the Assistant Professor of distributed and networked systems with the Faculty of Mathematics, Ruhr University Bochum, Germany. After graduation, he was a Carl-Zeiss Fellow with TU Kaiserslautern, a Research Fellow with the School of Computing at National University of Singapore and an ERCIM Fellow in the Department of Information Security and Communication Technology with NTNU Trondheim, Norway. His research interests include performance modeling and analysis of communication networks.

A.1.8 Tightening Network Calculus Delay Bounds by Predicting Flow Prolongations in the FIFO Analysis

This work was published in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2021 [75].

Tightening Network Calculus Delay Bounds by Predicting Flow Prolongations in the FIFO Analysis

Fabien Geyer[*†] Alexander Scheffler[‡] Steffen Bondorf[‡]

[*]Technical University of Munich
Munich, Germany

[†]Airbus Central R&T
Munich, Germany

[‡]Faculty of Mathematics, Center of Computer Science
Ruhr University Bochum, Germany

***Abstract*—Network calculus offers the means to compute worst-case traversal times based on interpreting a system as a queueing network. A major strength of network calculus is its strict separation of modeling and analysis frameworks. That is, a model is purely descriptive and can be put into multiple different analyses to derive a data flow's worst-case traversal time bound. One of the recent results in this category is the so-called flow prolongation. Flow prolongation actively manipulates the internal model of the analysis by virtually extending the path of flows, i.e., by deliberately creating a more pessimistic setting of resource contention between flows. It was shown that flow prolongation can theoretically decrease worst-case traversal time bounds under certain assumptions. Yet, due to its exhaustive search, it was also shown that flow prolongation does not scale and it might not even have an impact in larger queueing networks. In this paper we introduce DeepFP, an approach to make the analysis scale by predicting flow prolongations using a graph neural network. In our evaluation, we show that DeepFP can improve results in networks of FIFO queues considerably, where the delay bound can be reduced by 13.7 % in large FIFO networks at negligible additional cost on the execution time of the analysis.**

I. Introduction

Nowadays, many newly developed networked systems aim to provide some kind of performance guarantee – prime examples are those in the automotive and avionics sector [1] as well as factory automation [2]. Applications in these domains that rely on network performance care about one important property: the worst-case traversal time, i.e., the end-to-end delay, of data communication. Safety-critical applications that are crucial for the entire system's certification even need to show guaranteed upper bounds on the end-to-end delay.

Network Calculus (NC) offers a framework for this purpose. It consists of two parts: modeling and analysis. For best results, i.e., tight delay bounds, both should be developed in lockstep to prevent mismatches in their capabilities. Unfortunately, this has not always been the case and the analysis needs to catch up. Some easy to model network characteristics such as multicast flows [3] or ring topologies [4, 5] have only seen more detailed treatment recently. Thanks to the analysis' independence of the descriptive model, other characteristics also found their way into the analysis. Most prominent are the properties Pay Bursts Only Once (PBOO) [6] and Pay Multiplexing Only Once (PMOO) [7] that prevent the analysis from assuming data flows to exhibit stop-and-go behavior and/or overtake each other multiple times when crossing a sequence of servers (so-called tandems). In general, improvements to the NC tandem analysis tried to remove pessimistic assumptions

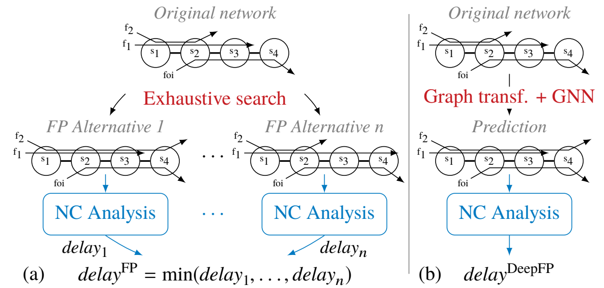


Figure 1: Comparison between the (a) original FP [11] with $O(n^m)$ NC analyses and (b) our DeepFP with one prediction.

from its internal model in order to improve the derived delay bounds for the original, user-provided model. Unfortunately, this also lead to the situation that none of the fast, algebraic NC analyses is strictly best. A search for the most suitable analysis is often advised [8, 9, 10]. Novel analysis features try to narrow down the amount of potentially best analyses.

An entirely different approach was recently presented with the Flow Prolongation (FP) feature [11]. It actively converts the network model given to the NC analysis to a more pessimistic one that circumvents limitations of the NC analysis capabilities. The analysis derives algebraic NC terms bounding a flow's delay. The amount of terms grows exponentially with the network size and none of them computes the tightest bound for all data flow descriptions. All need to be derived and solved [9]. The analysis has to compute a multitude of valid delay bounds to find the minimum among them. FP not only increases the amount of algebraic terms (and thus bounds), it also complicates the prediction of a term's added pessimism.

FP is conceptually straight-forward: assume cross-flows take more hops than they actually do. Nonetheless FP is a powerful feature to add to a NC analysis, it was even adopted in the Stochastic Network Calculus [12]. Unfortunately, finding the best prolongation alternative is prone to a combinatorial explosion. On each tandem of length n with m cross-flows, there are $O(n^m)$ alternatives to prolong flows. Even with a deep understanding of the NC analysis applied to reduce FP alternatives it could not be made to scale to larger models [11].

A novel approach to overcome exhaustive searches in the algebraic NC analysis was recently proposed: Graph Neural Network (GNN) predictions for NC term creation. This can

be used to make the NC analysis scale by restricting the exhaustive search to few alternatives [13, 14]. We base our contribution on this work, presenting *DeepFP* illustrated in Figure 1.

By demonstrating that we can make the FP analysis scale this way, we also reveal that its impact on the derived delay bound is very sensitive to the network model's assumptions. The foremost contribution of this paper is the FP analysis of FIFO networks. Under this assumption and applying the state-of-the-art algebraic Least Upper Delay Bound (LUDB) analysis [15, 16], we derive entirely new conditions for beneficial flow prolongations, train the GNN and acquire significantly improved delay bounds.

Our results can be applied to any system designed around FIFO-multiplexing and -forwarding of data. Most notable are Ethernet-based networks like Avionics Full-DupleX Ethernet (AFDX) or IEEE Time-Sentitive Networking (TSN). Even though they follow the "FIFO per priority queue" design, their NC model is essentially a FIFO system model. Our results can be combined with existing works on service modeling of the specific schedulers used in those systems.

This paper is organized as follows: Section II presents the related work and Section III gives an overview on NC analyses. In Section IV, we show how FP can improve bounds in FIFO multiplexing networks as well as the challenge it imposes. Section V provides the DeepFP method to make FP applicable to a wide range of networks. Section VI evaluates DeepFP and Section VII concludes the paper.

II. Related work

NC and RTC: Network Calculus takes a purely descriptive model of a network of queueing locations and data flows (see Appendix A). The NC analysis then computes a bound on the worst-case delay for a certain flow, the flow of interest (foi) (see Appendix B). A variant of NC that focuses on (embedded) real-time systems is the so-called Real-Time Calculus (RTC) [17]. Equivalence between the slightly differing resource descriptions has been proven in [18]. What remains is the difference in modeling of the "network" and the analysis thereof. RTC models networks of components such as the Greedy Processing Component (GPC) [19, 20]. Each component represents a macro, i.e., a fixed sequence of algebraic NC operations to apply to its input. Thus, the model already encodes the analysis. Moreover, this component modeling mostly restricts the analysis to strict priority multiplexing, yet, efforts to incorporate advanced properties such as PBOO and PMOO can be found in the literature [21, 22]. We, in contrast, aim for a model-independent improvement of the automatic derivation of a valid order of NC operations – the process called NC analysis – for networks of FIFO multiplexing systems. First results on this topic in NC [6, 23] were refined to the LUDB analysis [15, 16]. Later works entirely replace the algebraic NC analysis with an optimization one [24, 25], a mixed integer linear programming formulation that introduces forbiddingly large computational effort. Current efforts try to improve it by trading off delay bound tightness [26].

Flow Prolongation (FP): We pair FP with LUDB's DEBORAH tool to counteract its main tightness-compromising problem, thus considerably improving delay bounds. There has been one previous mention of FP in FIFO networks: [15] briefly shares the observation that, if prolonged, a cross-flow can be aggregated with the foi – independent of the LUDB problem we tackle. This can be combined with our contribution. We leave its investigation to future work.

Prolonging at the front may also be possible, but only in the arbitrary multiplexing PMOO analysis [27].

Graph Neural Networks: GNNs were first introduced in [28, 29] and [30] presents a framework that formalizes many concepts applied in GNNs in a unified way. GNNs were already proposed as an efficient method for speeding up exhaustive searches or similar NP-hard problems such as the traveling salesman problem [31]. A recent survey [32] about existing applications of machine learning to formal verification shows that this combination can accelerate formal methods, e.g., theorem proving, model-checking, Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) problems.

For computer networks, they have recently been applied to prediction of average queuing delay [33] and different non-NC performance evaluations of networks [34, 35, 36, 37]. [38] recently used GNNs for predicting the feasibility of scheduling configurations in Ethernet networks.

NC and GNN: DeepTMA was proposed in [13, 39] as a framework where GNNs were used for predicting the best contention model to use whenever there are alternatives for a tandem. DeepFP and DeepTMA are closely related: both methods use a graph transformation and a GNN to replace a computationally expensive exhaustive search. While DeepTMA targeted the Tandem Matching Analysis (TMA) [9], DeepFP focuses on FP and therefore we need to design a different graph transformation to connect NC and the GNN. Moreover, DeepTMA was shown to scale to large networks with up to 14 000 flows [39] in follow-up improvements of the method.

III. Network Calculus Analyses

The main objective of NC is to derive a bound on the flow of interst's (foi's) end-to-end delay, subject to interference and queuing. The resulting order of data in a shared queue when two different flows multiplex is a main concern of the NC analysis. NC generally differentiates between no assumption at all, so-called arbitrary multiplexing, and FIFO multiplexing. Given curves β lower bounding available forwarding service and α upper bounding arriving data (see Appendix A), NC can compute lower bounds on a foi's residual service.

Theorem 1 (Residual Service Curve): Consider a server s that offers a strict service curve β. Assume flows f_1 and f_2 with arrival curves α_1 and α_2, respectively, traverse the server. We can compute the service curve for guaranteed residual service for f_1, subject to multiplexing of flows at s, as

$$\beta^1(t) = [\beta(t) - \alpha_2(t)]^{\uparrow} =: \beta \ominus \alpha_2 \quad (1)$$

for arbitrary multiplexing and as

$$\beta^1(t,\theta) = [\beta(t) - \alpha_2(t-\theta)]^{\uparrow} \cdot \mathbf{1}_{\{t>\theta\}} =: \beta \ominus_\theta \alpha_2, \forall \theta \geq 0 \quad (2)$$

for FIFO multiplexing [40, Theorem 4]. $\mathbf{1}_{\{\text{condition}\}}$ denotes the indicator function (1 if the condition is true, 0 otherwise) and $[g(x)]^{\uparrow} = \sup_{0 \le z \le x} g(z)$ is the non-decreasing closure of function $g(x)$ defined on positive real values.

In a FIFO multiplexing server, the residual service depends on the flow of interest (f_1 in Theorem 1). Yet, it is desired to be computed seemingly independent of it as with arbitrary multiplexing. θ encodes the FIFO worst cases for any foi. It thus defines an infinite set of (valid) residual service curves.

A. PMOO, the Analysis for Arbitrary Multiplexing

An important discovery in the evolution of analysis capabilities was that, even assuming arbitrary multiplexing, cross-flows' worst-case burstiness need not be assumed to fully collide with the foi at each server. This is known as Pay Multiplexing Only Once (PMOO) [7]. To achieve this, the proposed PMOO analysis computes a residual service curve for an entire tandem of servers.

The PMOO analysis has a disadvantage when analyzing feed-forward networks. To handle demultiplexing on the foi's path, cross-flows interfering on different foi-subpaths need to be analyzed in a demultiplexed fashion in the entire feed-forward analysis. At shared servers before the foi's path, that creates mutually exclusive worst-case assumptions [41]. For an example, see Figure 2 where at server s_1 each cross-flow, f_1 and f_2, would compute a residual service curve to demultiplex from the other.

B. DEBORAH, the Analysis Tool for FIFO Multiplexing

For the analysis of FIFO multiplexing tandems, there are two challenges:
a) implementing the PMOO principle and
b) finding the best setting for the free θ parameter in the residual service curve computation.

The Least Upper Delay Bound (LUDB) analysis [42, 15, 16] tackles both. As its name suggests, *b)* is achieved by finding the smallest among many alternative delay bounds (similar to Figure 1(a)). To do so, LUDB converts the problem of setting all θ in the algebraic NC term into several linear programs. This conversion strictly requires the modeling curves to be affine, a restriction we inherit in this paper.

The more important part for our flow prolongation is the current solution to challenge *a)*. LUDB does not necessarily achieve a full implementation of the PMOO principle. It is very susceptible to the nesting of flows on the analyzed tandem. In general, a tandem is called nested if any two flows have disjunct paths or one flow is completely included in the path of the other flow. For example, in Figure 2(a), f_2 is completely included in the path of f_1 but neither are completely included in the foi's path. If flows form a non-nested interference pattern as f_1 and f_2 in Figure 3(a), then LUDB needs to cut the tandem into a sequence of sub-tandems, each with a nested interference pattern. At these cuts, a tighness-reducing computation to bound the arrivals of cross-flows needs to be executed. It adds the cross-flow burstiness to all sub-tandem residual service curve computations and

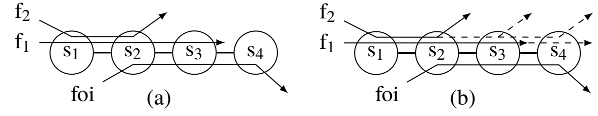


Figure 2: (a) Example tandem network shown in Figure 1 and (b) indication of all its potential flow prolongation alternatives

PMOO is not achieved. Therefore, we aim at reducing the amount of cuts.

Last, the DEBORAH tool has been developed to implement LUDB [43] for tandem networks only. We extended it to analyze feed-forward networks in our numerical evaluations.

C. Flow Prolongation

Flow Prolongation was designed as an add-on feature to mitigate the PMOO analysis's problem described above [11]. FP is, however, a generic approach that is independent of any multiplexing assumption. More formally, it is defined by:

Corollary 1 (Delay Increase due to FP): Assume a tandem $\mathcal{T}$ defined by the foi's path. Let the foi be f_1 and let there be cross-flows on $\mathcal{T}$. A prolongation of cross-flows to create tandem $\mathcal{T}_{\text{FP}}$ increases the end-to-end delay of f_1 on $\mathcal{T}_{\text{FP}}$.

Proof 1: Wlog assume a single cross-flow f_2 to be prolonged over one additional server s where f_1 is present, too. Compared to $\mathcal{T}$, s in $\mathcal{T}_{\text{FP}}$ multiplexes incoming data of f_2 with data of f_1 in its queue. s either forwards this data of f_2 after f_1, causing no increase of f_1's delay on $\mathcal{T}_{\text{FP}}$, or it forwards at least parts of the data of f_2 before f_1, causing an additional queuing delay to f_1.

Corollary 1 shows that FP is a conservative transformation adding pessimism to the network model that increases the foi's delay. For delay bounds, it holds that:

Corollary 2 (FP Delay Bound Validity): Assume a tandem $\mathcal{T}$ defined by the flow of interest's path. Let $\mathcal{T}_{\text{FP}}$ be derived from $\mathcal{T}$ by flow prolongation. Then, the bound on the foi's worst-case delay in $\mathcal{T}_{\text{FP}}$ is a bound on the foi's delay on $\mathcal{T}$.

Proof 2: Per Corollary 1, we know that the foi's end-to-end delay will not decrease by FP. Thus, the tight delay bound in $\mathcal{T}_{\text{FP}}$ will exceed the tight delay bound on $\mathcal{T}$ and any potentially untight bound derived for $\mathcal{T}_{\text{FP}}$ bounds the foi's delay on $\mathcal{T}$.

Take the sample tandem in Figure 2(a), where bounding the arrivals of data flows f_1 and f_2 is required at their first location of interference with the foi, server s_2. Assuming arbitrary multiplexing, the PMOO analysis suffers from the segregation effect [41], both flows assume to only receive service after the respective other flow was forwarded by server s_1 – an unattainable pessimistic forwarding scenario in the analysis-internal view on the network. FP tries to steer the analysis such that it does not have to apply this pessimism by prolonging flows inside the analysis: the dashed lines in Figure 2(b), depict potential prolongations of the two flows' paths. Each prolongation alternative that matches their sinks will allow for their aggregate treatment at s_1, mitigating the problem. Yet, this adds interference to the foi. Therefore, we search for the best prolongation alternative trading off both

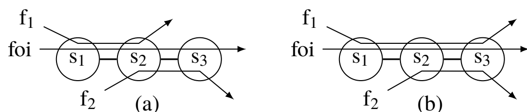


Figure 3: (a) Tandem network and (b) its prolonged version

aspects. This search approach does not scale, neither are there hopes that PMOO delay bounds improve much [11].

IV. Flow Prolongation in the NC FIFO Analyses

In this Section, we address the question of how flow prolongation can improve the NC-derived worst-case delay bound for a flow of interest in the FIFO analysis.

In the PMOO analysis, demultiplexing is the dominant problem that causes a loss of tightness. While the problem of demultiplexing applies to the LUDB analysis for FIFO networks, too, it suffers from yet another and more impactful problem that we address with flow prolongation: the lack of the PMOO property. To implement the property, an analysis needs to first create an end-to-end view on a tandem. LUDB, and thus the DEBORAH tool, cannot achieve this for non-nested interference patterns (see Figure 3(a)). It can only analyze nested tandems in a PMOO fashion where cross-flow paths do not overlap.

To apply LUDB nonetheless, the tandem is cut into a sequence of sub-tandems with nested interference patterns. In Figure 3, the tandem can be cut before or after server s_2. Either alternative has the very same drawback: a cross-flow is cut, too, and to get it onto the subsequent sub-tandem, an explicit bound on its arrivals has to be computed. This is achieved with the deconvolution $\oslash$ (see Appendix B) or Theorem 2 in Section IV-A, adding the cross-flow's original burst term to the analysis once more – PMOO is not achieved. Let server s_i provide service β_i and let flow f_j put α_j data into the network. The respective (min,plus)-analysis terms using $\oslash$ as derived by DEBORAH which bound the foi's delay are:

$$h(\alpha_{\text{foi}}, (\beta_1 \ominus_\theta \alpha_1) \otimes (((\beta_2 \ominus_\theta (\alpha_1 \oslash (\beta_1 \ominus_\theta \alpha_{\text{foi}}))) \otimes \beta_3) \ominus_\theta \alpha_2)) \tag{3}$$

for the cut left of s_2 and for the cut right to it:

$$h(\alpha_{\text{foi}}, ((\beta_1 \otimes (\beta_2 \ominus_\theta \alpha_2)) \ominus_\theta \alpha_1) \otimes (\beta_3 \ominus_\theta (\alpha_2 \oslash (\beta_2 \ominus_\theta ((\alpha_{\text{foi}} + \alpha_1) \oslash \beta_1))))). \tag{4}$$

Curves and binary (min,plus)-operations are defined in Appendix A. For this example, it is already sufficient to note that every occurrence of the deconvolution $\oslash$ reduces the tightness of the computed delay bound.

In this paper, we devise an alternative strategy to create a tandem with nested interference only and thus less cuts, less occurrences of $\oslash$ and more PMOO property implementation: flow prolongation. By prolonging cross-flow f_1 in this small sample tandem by another hop, we create the one shown in Figure 3(b). Without overlapping interference, the foi's DEBORAH-derived delay bound becomes:

$$h(\alpha_{\text{foi}} + \alpha_1, \beta_1 \otimes ((\beta_2 \otimes \beta_3) \ominus_\theta \alpha_2)) \tag{5}$$

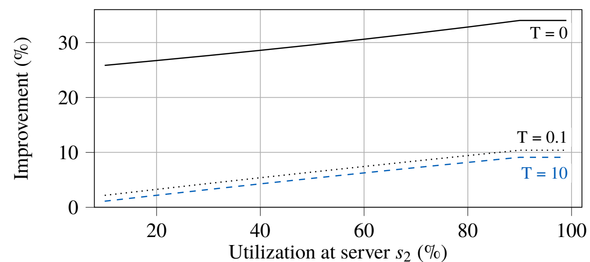


Figure 4: Delay bound improvements by using FP in the DEBORAH analysis of the network in Figure 3

By its lack of deconvolutions, i.e., single appearances of each involved flow's arrival curve, the term clearly shows that the PMOO principle is implemented. Yet, at the expense of aggregating the foi with its cross-flow f_1[1]. We have tested this new instantiation of flow prolongation to improve the DEBORAH-derived LUDB bounds for different curve settings in the network shown in Figure 3. Service curves were set to $\beta_{R=30,T}$ and arrival curves to $\gamma_{r=\frac{u}{10},0.1}$ where u denotes the utilization $\frac{3r}{R}$ at the server that always sees three flows, s_2, and varying latencies. Note, that our setting guarantees for finite delay bounds. Figure 4 shows the results.

FP for the DEBORAH analysis, henceforth called DEBORAH-FP, is a very promising approach to implement the PMOO property in the algebraic NC analysis. Its application vastly differs from the PMOO analysis in arbitrary multiplexing. Put simple, the necessary preconditions for FP to have a positive impact on each analysis are as follows:

- For the PMOO analysis, prolong cross-flows that start at *the same* server to the same last server.
- For the DEBORAH analysis, prolong cross-flows that start at *different* servers to the same last server.

A. DEBORAH in feed-forward FIFO networks

For the analysis of feed-forward FIFO networks, we integrated DEBORAH into the NetworkCalculus.org Deterministic Network Calculator (NCorg DNC) [44] as its feed-forward analysis already provides the required decomposition of the network into a sequence of tandems [45]. Second, LUDB only computes delay bounds but we can use DEBORAH for bounding arrivals of cross-traffic by using the following theorem, an alternative to the deconvolution-based computation:

Theorem 2 (Output From Delay [46]): Consider a tandem of servers $\mathcal{T}$ that offers a service curve β. Assume flow f with arrival curve α traverses $\mathcal{T}$, experiencing a delay bounded by d. Then $\alpha'(t) = \alpha(t + d)$ bounds the output of f from $\mathcal{T}$.

The impact of our contribution does not rely on this rather inaccurate bounding technique. We put flow prolonged

[1]DEBORAH can only work with a single flow (aggregate) per distinct path on the tandem. Input to DEBORAH needs to be formatted accordingly. In case a cross-flow has the same path as the foi we thus get an aggregate delay bound instead of computing a residual service curve for the foi – an alternative derivation of a valid upper delay bound that now implements PMOO, too. Either is subject to overly pessimistic interference assumptions.

tandems into the DEBORAH tool that applies a more refined computation internally if the tandem is non-nested. Yet, DEBORAH does not expose this computation to the user.

A recent overview on further NC tools can be found in [47]. We also investigated another tool[2] for the analysis of FIFO networks, which uses a linear program (LP) to compute the delay bound. Our evaluations showed that it scaled insufficiently for inclusion in our numerical evaluation, even on small networks with 20 flows, mainly due to the large number of LP constraints generated, confirming previous results [9].

B. The Challenge to Apply Flow Prolongation

As mentioned in Section I and illustrated in Figures 1(a) and 2, on each tandem of length n with m cross-flows, FP may explore $O(n^m)$ prolongation alternatives. It was shown for arbitrary multiplexing that exhaustive FP analysis does not scale in feed-forward networks [11]. Due to their similarity, the scaling problem also holds when applying DEBORAH-FP.

1) Restricting the Application of FP: The most straightforward trade-off between delay bound tightness and computational complexity is, of course, to restrict the use of FP inside the NC analysis. We deviate from an exhaustive use of FP on every tandem to a selective use where it has the most impact on the delay bound. It turned out that this is achieved by only applying FP to the analysis of the foi, not for bounding the arrivals of its cross-flows. This creates the FP_{foi} variants PMOO-FP_{foi} and DEBORAH-FP_{foi}. Figure 5 illustrates the delay bound gap between PMOO-FP and PMOO-FP_{foi}, and between DEBORAH-FP and DEBORAH-FP_{foi}, namely:

$$delay\ bound\ gap = \frac{delay_{\text{foi}}^{\text{FP}_{\text{foi}}} - delay_{\text{foi}}^{\text{FP}}}{delay_{\text{foi}}^{\text{FP}}} \tag{6}$$

For more than 99 % of the studied flows, the delay bound is unchanged. On average, the relative error is only of 0.58 % for PMOO and 1.18 % for DEBORAH. Those values illustrate that the loss of tightness of using FP_{foi} instead of FP is minimal.

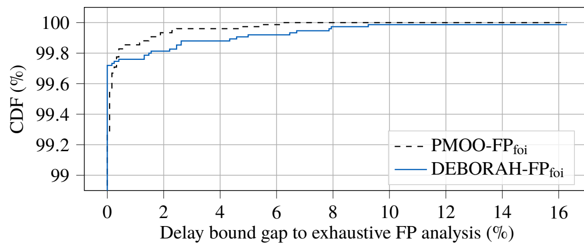


Figure 5: Delay bound gap of FP_{foi} analyses based on the evaluation dataset presented in Section V-D

In order to see the impact on the execution time of running flow prolongations only on the foi's analysis, Figure 6 illustrates the relative execution time of FP against FP_{foi}, namely:

$$Relative\ execution\ time = \frac{Execution\ time\ FP}{Execution\ time\ FP_{foi}} \tag{7}$$

[2]https://github.com/annebouillard/NetCalBounds based on [24, 25]

PMOO-FP_{foi} is 1.3 times faster than PMOO-FP in average, while there is almost no difference in execution time between DEBORAH-FP and DEBORAH-FP_{foi}.

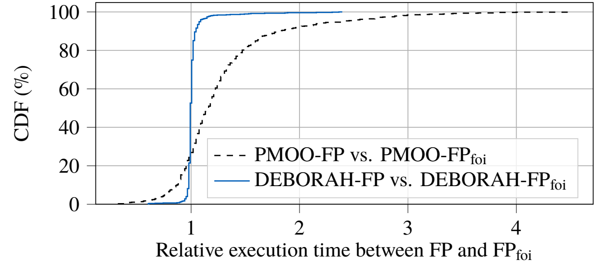


Figure 6: Relative execution time of a flow's analysis based on the evaluation dataset presented in Section V-D

2) PMOO-FP's explored alternatives: We can reasonably reduce the use of FP to a single tandem. On this tandem, the amount of prolongation alternatives to explore can be further reduced. In practice, not all cross-flows go over only the first server such that they can also be prolonged to any following one. Moreover, PMOO-FP already cuts out all those alternatives that cannot impact the analysis by circumventing the need to carry over demultiplexing – as described in the necessary FP precondition above. Similarly, we improved DEBORAH-FP to not prolong if there is no potential to convert a non-nested interference pattern to a nested one. Still, the amount of prolongation alternatives for the dataset evaluated in this paper is forbiddingly large, see Figure 7. Note for a large number of cross-flows, networks may have been excluded from this preliminary evaluation due to a 1 hour deadline set for computing data. As expected, we get an exponential scaling between the number of cross-flows and the number of explored alternatives.

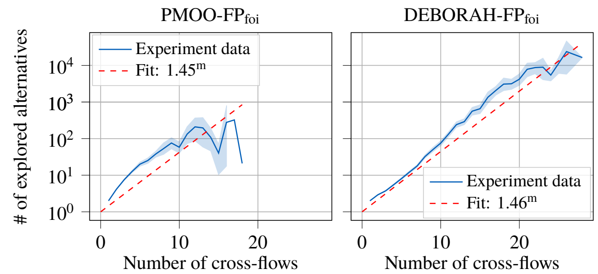


Figure 7: Relation between the number of cross-flows and the number of explored prolongation alternatives by PMOO-FP_{foi} and DEBORAH-FP_{foi}

Overall, we need a better way to find the best prolongation alternative that improves the delay bound to be derived. In this paper, we propose DeepFP that can be trained on either a PMOO-FP_{foi} or a DEBORAH-FP_{foi} dataset to predict the best alternative(s). We show that DeepFP makes the FP feature scale, that PMOO-FP_{foi} provides only minor improvements

over PMOO and that, in contrast, FP has a considerable impact on the LUDB analysis when coupled with its implementation, DEBORAH, to DEBORAH-FP$_{\text{foi}}$.

V. Effective FP Predictions with a GNN

We make the FP analysis scale with GNN predictions and show that the impact vastly depends on the multiplexing assumption. We develop our universal DeepFP heuristic in this section, based in part on the work proposed in DeepTMA [13, 14]. As illustrated in Figure 1 and Algorithm 1, the main intuition behind DeepFP is to avoid the exhaustive search for the best prolongation by limiting it to a few alternatives. The heuristic's task is then only to predict the best flow prolongations, which are then fed to the NC analysis. This ensures that the bounds provided are formally valid.

Algorithm 1 DeepFP analysis of network $\mathcal{N}$ and flow f_{foi}

$\mathcal{G}$:= graphTransformation($\mathcal{N}$, f_{foi}) → see Algorithm 2
$prolongations$:= GNN($\mathcal{G}$) → see Section V-A
$\mathcal{N}_p$:= networkWithFlowProlongations($\mathcal{N}$, $prolongations$)
return Network Calculus analysis of $\mathcal{N}_p$ and f_{foi}

For DeepFP, we used a Graph Neural Network (GNN) as heuristic, since it was shown in DeepTMA to be a fast and efficient method. We define networks to be in the NC modeling domain and to consist of servers, crossed by flows. We refer to the model used in GNN as graphs. Our heuristic transforms the networks into graphs, which are processed by the GNN. The output of the GNN is then fed to PMOO-FP$_{\text{foi}}$ or DEBORAH-FP$_{\text{foi}}$, which finally performs the NC analysis on the subset of combinations suggested by the GNN.

A. Graph Neural Networks

As for DeepTMA, we use the framework of GNNs introduced in [28, 29]. They are a special class of neural networks for processing graphs and predict values for nodes or edges depending on the connections between nodes and their properties. The idea behind GNNs is called *message passing*, where so-called *messages* – i.e., vectors of numbers $\mathbf{h}_v \in \mathbb{R}^k$ – are iteratively updated and passed between neighboring nodes. Those messages are propagated throughout the graph using multiple iterations. We refer to [48] for a formalization of many concepts recently developed around GNNs.

As with DeepTMA, we selected Gated Graph Neural Networks (GGNN) [49] for our model, with the addition of edge attention. For the edge attention mechanism, we selected an approach similar to [50], where each edge (u, v) in the input graph is weighted with a parameter $\lambda_{(u,v)} \in (0, 1)$, such that:

$$\lambda_{(u,v)}^{(t)} = \sigma\left(FFNN\left(\left\{\mathbf{h}_u^{(t)}, \mathbf{h}_v^{(t)}\right\}\right)\right) \tag{8}$$

with $FFNN$ a feed-forward neural network $\mathbf{h}_v^{(t)}$ representing the message from node v at iteration t, σ the sigmoid function, and $\{\cdot, \cdot\}$ the concatenation. In summary, the hidden node update function becomes:

$$\mathbf{h}_v^{(t)} = GRU\left(\mathbf{h}_v^{(t-1)}, \sum_{u \in \text{NBR}(v)} \lambda_{(u,v)}^{(t-1)} \mathbf{h}_u^{(t-1)}\right) \tag{9}$$

with $\text{NBR}(v)$ of v the set of neighbors of node v, and GRU a Gated Recurrent Unit (GRU) [51].

B. Model transformation

Since we work with a machine learning method, we need an efficient data structure for describing a NC network which can be processed by a neural network. We chose undirected graphs, as they are a natural structure for describes networks and flows. Due to their dynamic sizes, networks of any sizes may be analyzed using our method.

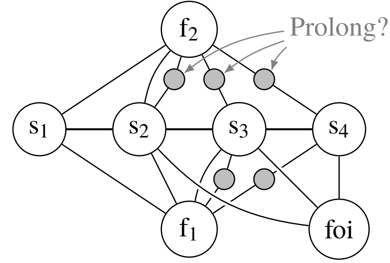


Figure 8: Graph encoding of the network from Figure 2(a)

We follow Algorithm 2 for this graph transformation, also illustrated and applied in Figure 8 on the network from Figure 2(a). Each server is represented as a node in the graph, with edges corresponding to the network's links. The features of a server node are its service curve parameters, namely its rate and latency. Each flow is represented as a node in the graph, too. The features of a flow node are its arrival curve parameters, namely its rate and burst. Additionally, the foi receives an extra feature representing the fact that it is the analyzed flow.

Algorithm 2 Graph transformation of network $\mathcal{N}$ for flow f_{foi}

$\mathcal{G}$:= empty undirected graph
for all server s_i **in** network $\mathcal{N}$ **do** $\mathcal{G}$.addNode(s_i)
for all link (s_i, s_j) **in** network $\mathcal{N}$ **do** $\mathcal{G}$.addEdge(s_i, s_j)
for all flow f_i **in** network $\mathcal{N}$ **do**
  $\mathcal{G}$.addNode(f_i)
  for all server s_j **in** f_i.path() **do** $\mathcal{G}$.addEdge(f_i, s_j)
for all flow f_i **in** network $\mathcal{N}$ excluding f_{foi} **do**
  for all server s_j **in** f_{foi}.path() **do**
    if prolongation $P_{f_i}^{s_j}$ of flow f_i to s_j is valid **then**
      $\mathcal{G}$.addNode($P_{f_i}^{s_j}$)
      $\mathcal{G}$.addEdges((f_i, $P_{f_i}^{s_j}$), ($P_{f_i}^{s_j}$, s_j))
return $\mathcal{G}$

To encode the path taken by a flow in this graph, we use edges to connect the flow to the servers it traverses. Compared

to the original DeepTMA graph model from [13], we simplify one aspect: we do not include path ordering nodes that tell us the order of servers on a crossed tandem. DeepTMA was shown to benefit only marginally from the effort to incorporate this additional information [14] and we confirmed the same behavior in preliminary DeepFP numerical evaluations.

To represent the flow prolongations, *prolongation* nodes ($P_{f_i}^{s_j}$) connecting the cross-flows to their potential prolongation sinks are added to the graph. Those nodes contain the hop count according to the foi's path as main feature – this is sufficient to later feed the prolongation into the NC analysis, path ordering nodes are not required for this step either.

The last server of a cross-flow's unprolonged path is also represented as a node (s_3 for f_1 and s_2 for f_2 in Figure 8). Those nodes represent the choice to not prolong a flow.

Based on this graph representation, we define two classification problems for the neural network. The first one is decide if it is worthwhile to apply the prolongation algorithm or not. For this, we use a binary classification of the foi node.

The second classification problem is to decide where to prolong the flows if necessary, by applying a binary classification on the prolongation nodes. Namely for each cross-flow f and each potential sink s, the neural network assigns a score $P_{f,s}$ between 0 and 1 to the corresponding prolongation node. For each flow, the prolongation node with the highest score decides which sink to use for prolonging the flow. As illustrated in Figure 1(b), those predictions are then fed to PMOO-FP$_{\text{foi}}$ or DEBORAH-FP$_{\text{foi}}$, which finally performs the NC analysis. Since the GNN might also choose not to prolong, the standard PMOO or DEBORAH analyses are only performed if explicitly requested by the GNN.

C. Implementation

We implemented the GNN used in DeepFP using PyTorch [52] and pytorch-geometric [53]. Optimal parameters for the neural network size and the parameters for training were found using hyper-parameter optimization. Table I illustrates the size of the GNN used for the evaluation in Section VI.

Layer	NN Type	Size
init	FFNN	$(11, 96)_w + (96)_b$
Memory unit	GRU cell	$(96, 96)_w + 2 \times \{(288, 96)_w + (96)_b\}$
Edge attention	FFNN	$(192, 96)_w + (96)_b + (192, 96)_w + (1)_b$
out hidden layers	FFNN	$2 \times \{(96, 96)_w + (96)_b\}$
out final layer 1	FFNN	$(96, 1)_w + (1)_b$
out final layer 2	FFNN	$(96, 1)_w + (1)_b$
	Total:	104 455 parameters

Table I: Size of the GNN used in Section VI. Indexes represent respectively the weights (w) and biases (b) matrices

D. Dataset generation

To train our neural network architecture using a supervised learning method, we generated a set of random tandem topologies (as to check the FP preconditions of Section IV). For each created server, a rate-latency service curve was generated with uniformly random rate and latency parameters. A random number of flows was generated with random source and sink servers. For each flow, a token-bucket arrival curve was generated with uniformly random burst and rate parameters. All curve parameters were normalized to the $(0, 1]$ interval.

For each generated topology, the NCorg DNC v2.6.1 [44] is then used for analyzing each flow and record the different iterations of PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$. Namely we extract the combinations of flow prolongations which resulted in the lowest end-to-end delay during the exhaustive search. Each analysis is run with a maximum deadline of 1 hour.

We extended the NCorg DNC tool to integrate the DEBORAH tool. For our evaluations, we run DEBORAH in so-called STA mode (Single Tandem Analysis) [16] instead of the default MSA (Multiple Sub-tandem Analysis). The MSA mode computes per-sub-tandem delay bounds and adds them up. In this mode, DEBORAH cannot be used to implement the PMOO principle, not even the PBOO one. Yet, STA has a worse execution time since more variables have to be optimized simultaneously.

Since PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$ may not bring any benefits compared to PMOO or DEBORAH, either due to no alternatives for prolonging flows or no end-to-end delay improvement by any alternative, we restrict the dataset to networks and flows where FP is applicable (i.e., flows with prolongation options). Table II contains statistics about the generated dataset. In total approximately 54 000 flows were generated and evaluated for the training dataset, and 10 000 for the numerical evaluation presented in Section VI. The dataset is available online[3] to reproduce our learning results.

Parameter	Min	Max	Mean
# of servers	4	10	7.8
# of flows	5	35	24.5
# of cross-flows	1	21	4.1
# of prolong. comb. (PMOO-FP$_{\text{foi}}$)	2	4024	16.8
# of prolong. comb. (DEBORAH-FP$_{\text{foi}}$)	2	131072	247.1
Flow path length	3	9	4.1
Number of nodes in graph	11	128	43.3

Table II: Statistics about the generated dataset

E. Neural network training

We use standard gradient descent techniques to train our GNN, using the binary cross-entropy loss function, namely the optimization goal is to minimize:

$$loss(T, P) = \sum_{f \in \mathcal{F}, s \in \mathcal{S}_f} \left(T_{f,s} \log P_{f,s} + (1 - T_{f,s}) \log (1 - P_{f,s})\right) \tag{10}$$

with T_f^s representing the target score for the prolongation, with 1 if it is selected for prolongation and 0 otherwise.

We follow a standard supervised learning approach for training the neural network. Since the choice of flow prolongations may have multiple equally-optimal solutions, the choice of which target solution to provide as training data for the neural network is not obvious. In other words, the target vector $T_{f,s}$ in Equation (10) has to be defined according to a single

[3] https://github.com/fabgeyer/dataset-rtas2021

solution, but multiple equally good solutions are available. In our experiments, training the GNN on a single solution resulted in poor convergence of the model.

To provide target vectors which enable the neural network to be trained efficiently, we use here a concept inspired by hindsight loss [54, 55]. We dynamically find the correct target vector T which is the closest to the predicted score by the neural network and use it in the loss function. The loss function introduced in Equation (10) becomes:

$$\mathcal{L} = \min_{T \in \tau_{opt}} loss(T, P) \tag{11}$$

where τ_{opt} is the set of flow prolongation choices leading to an optimal solution.

Since we address two FP instantiations that are even orthogonal as seen in their preconditions to have a positive impact, we define two versions of DeepFP as PMOO-DeepFP$_{\text{foi}}$ and DEBORAH-DeepFP$_{\text{foi}}$. The same graph representation and features are used regardless of the NC analyses, but two different training processes and resulting trained weights of the GNN are produced.

F. Flow prolongation choices

To improve the outcome of a DeepFP analysis at a small computational cost, we propose here to use the prediction vector of the GNN to generate multiple flow prolongation combinations. Those combinations are then analyzed using the NCorg DNC and the combination leading to the lowest end-to-end delay is kept. We name this extension DeepFP$_k$ where k corresponds to the number of combinations generated.

First, we consider the prediction vector of the GNN as a vector of probabilities of where to prolong flows. A categorical distribution parameterized by those probabilities is generated for each cross-flow and used to generate the k combinations.

This first version of DeepFP$_k$ does not make use of the expert knowledge mentioned in Section IV-B2, where some combinations are excluded from PMOO-FP$_{\text{foi}}$'s and DEBORAH-FP$_{\text{foi}}$'s exhaustive searches since they are known to be of lower quality than other combinations. In other words, the GNN can choose a combination of flow prolongations which might not have been explored by PMOO-FP$_{\text{foi}}$'s or DEBORAH-FP$_{\text{foi}}$'s exhaustive search. This reflects the generally observed wish to apply machine learning to a dataset without becoming an expert in the domain and without tailoring the dataset to learn from accordingly.

Last, we describe a second extension of DeepFP, called DeepFP$^+$ which is able to use this expert knowledge. In order to avoid selecting those excluded combinations, we define a matrix of explored combinations $\mathcal{C}$ containing their target vectors, with dimensions ($\#$ *of combinations*, $\#$ *of prolongation nodes*). Using the prediction vector $P_{f,s}$ from the GNN, we compute a vector containing a score for each combination:

$$CombinationScores = \mathcal{C} \times \left[P_{f_i,s_k} \cdots P_{f_j,s_l}\right]^T \tag{12}$$

To generate k combinations, we select the top-k combinations having the best scores in the $CombinationScores$ vector.

We numerically evaluate later in Sections VI-B and VI-D the impact of DeepFP$^+$ in terms of tightness and additional execution time of enumerating the combinations used by PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$.

VI. Numerical evaluation

Our numerical evaluation aims to answer two questions:

1) How much delay bound improvement can FP achieve?
2) How well does the GNN predict the FP alternative?

As FP does not scale well and we therefore proposed DeepFP in the first place, both aspects are naturally intertwined.

In the following, we show details about DeepFP performance in terms of tightness as well as execution time. Improvements in both will directly be applicable to and have an impact on any real-world application of the NC methodology. In order to illustrate the benefits of DeepFP, we also do a comparison against a heuristic which randomly selects one or multiple prolongation alternatives – a low-effort, non-expert alternative to add FP to an analysis. We label this heuristic as *RND*$_k$ in this section, with k being the number of random alternatives evaluated. At first, we use DeepFP without the extension using additional expert knowledge described in Section V-F. All evaluations presented here were done with the evaluation dataset described in Section V-D, except for Section VI-C which used larger networks.

A. Accuracy and delay bound gap

To quantitatively evaluate the performance of our approach, we use the relative gap between the delay bound given by PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$ and the delay bound given by a heuristic, incl. the non-FP original analysis:

$$delay\ bound\ gap_{\text{foi}}^{\text{FP}} = \frac{delay_{\text{foi}}^{\text{heuristic}} - delay_{\text{foi}}^{\text{FP}_{\text{foi}}}}{delay_{\text{foi}}^{\text{FP}_{\text{foi}}}} \tag{13}$$

A value of $delay\ bound\ gap_{\text{foi}}^{\text{FP}}$ close to zero indicates that the heuristic produced a tight result compared to the exhaustive search. Larger values indicate that the heuristic chose a bad prolongation, i.e. the bound is loose.

The results are shown in Figure 9. First to note is that FP does not have a significant impact in PMOO – we confirm the finding of [11] in a larger evaluation by observing an average gap between PMOO-FP$_{\text{foi}}$ and PMOO of just 3.7 %. Neither the random heuristic nor DeepFP can thus achieve a considerable delay bound improvement, although the predictions taken are very accurate.

For DEBORAH-FP, we can report a completely different picture. Having brought the FP property to the DEBORAH analysis had a huge impact on the delay bound tightness. We see that an average gap of 60.75 % between DEBORAH and DEBORAH-FP$_{\text{foi}}$ analysis results was opened when adding the exhaustive FP$_{\text{foi}}$ feature. Moreover, reducing the effort by random selection of prolongation alternatives did not perform well, even RND$_{16}$ leaves an average gap of 11.68 %. On the other hand, our DeepFP closes this gap successfully. Even the version with a single prediction pushes the gap down to 2.57 %

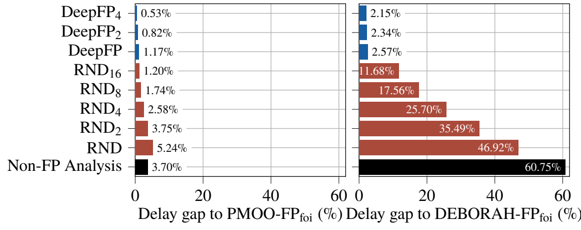


Figure 9: Average delay bound gap of heuristics against PMOO-FP$_{foi}$ and DEBORAH-FP$_{foi}$

such that an increase of proposed prolongation alternatives does not have a big impact anymore.

More detailed results are presented in Figure 10, where we illustrate the delay bound gap of DeepFP, the random heuristic and standard PMOO or DEBORAH analyses, confirming our findings that DEBORAH-FP$_{foi}$ is a big improvement over DEBORAH and DeepFP is the key to its efficient application.

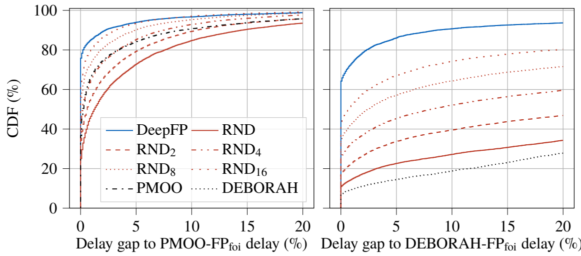


Figure 10: Delay bound gap of heuristics against PMOO-FP$_{foi}$ and DEBORAH-FP$_{foi}$

The accuracy of DeepFP is shown in Figure 11. For each analyzed flow in the test dataset, the method is accurate if the computed end-to-end delay bound is equal to the best end-to-end delay bound computed by PMOO-FP$_{foi}$ or DEBORAH-FP$_{foi}$. In average, DeepFP is able to predict the correct prolongation for 69.6 % of the flows for PMOO-FP$_{foi}$ and 60.9 % for DEBORAH-FP$_{foi}$. Generating multiple combinations as introduced in Section V-F increases the accuracy to 75.3 % and 64.3 % respectively for $k = 4$. In comparison, the random heuristic with one choice achieves only 14.5 % and 8.7 % respective accuracy. DeepFP is making more reasonable, more accurate predictions.

B. Impact of additional expert knowledge for DeepFP⁺

We introduced DeepFP⁺ in Section V-F, an extension of DeepFP making additional use of expert knowledge to explicitly filter out prolongation combinations which are known to be of lower quality. We numerically compare DeepFP and DeepFP⁺ in Figures 12 and 13. As expected, DeepFP⁺ is able to achieve a better accuracy for both PMOO-FP$_{foi}$ and DEBORAH-FP$_{foi}$. Nevertheless, while the delay bound gap of DeepFP⁺ to the exhaustive search of DEBORAH-FP$_{foi}$ is indeed reduced compared to DeepFP, DeepFP achieves better

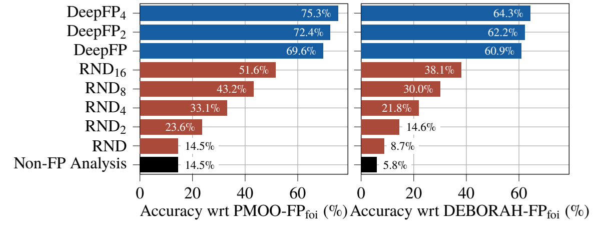


Figure 11: Accuracy of DeepFP, the random heuristic, and the non-FP analyses

Parameter	Min	Max	Mean
# of servers	2	16	8.7
# of flows	5	254	162.3
Flow path length	1	16	3.2

Table III: Statistics about the larger generated dataset

results than DeepFP⁺ for PMOO-FP$_{foi}$. This means that the expert knowledge of reducing the state of possible solutions might not be necessary.

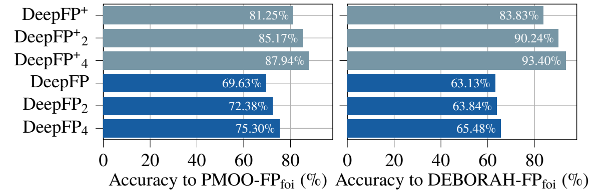


Figure 12: Accuracy of DeepFP⁺ and DeepFP

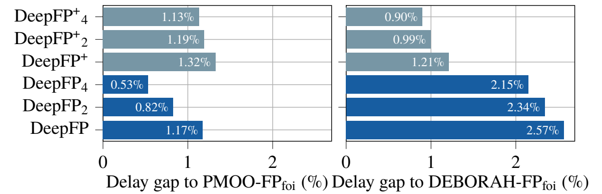


Figure 13: Average delay bound gap of DeepFP⁺ and DeepFP

The additional computational cost of using DeepFP⁺ will be evaluated later in Figure 17.

C. Scalability on larger networks

To evaluate the scalability of our approach with respect to the network size, we also evaluated DeepFP on networks with a larger number of servers and flows. The same random network generator as for the training dataset is used, but the number of servers and flows is scaled to larger values. Statistics about this additional dataset are presented in Table III. The training data used for the GNN is unchanged, namely we still restrict it to the smaller networks introduced in Section V-D and Table II.

The results of the exhaustive PMOO-FP$_{\text{foi}}$ and DEBORAH-FP$_{\text{foi}}$ are not available here due to their too long execution time, taking multiple days per network to compute in some cases. Instead, we use here the standard PMOO and DEBORAH analyses in order to evaluate the gain in tightness of using DeepFP. As in Equation (13), we define the delay bound gap to PMOO and DEBORAH (i.e. the analyses without the FP property) as:

$$\textit{delay bound gap}_{\text{foi}}^{\text{non-FP}} = \frac{\textit{delay}_{\text{foi}}^{\text{non-FP}} - \textit{delay}_{\text{foi}}^{\text{heuristic}}}{\textit{delay}_{\text{foi}}^{\text{non-FP}}} \quad (14)$$

A large positive value of $\textit{delay bound gap}_{\text{foi}}^{\text{non-FP}}$ indicates that the heuristic with the FP property gained tightness over the standard PMOO or DEBORAH analysis. In the opposite, a negative value indicates that the bound is less tight.

Numerical results are summarized in Figure 14. For the PMOO analysis, the random heuristic results in a negative delay bound gap in average, namely the resulting delay bounds are worse than by simply using the standard PMOO analysis, even for the larger values of $k = 32$. Despite this, DeepFP is able to achieve an average gain in tightness of 1.06 % for PMOO. For the DEBORAH analysis, the random heuristic results in a gain in tightness of only 0.25 %, where DeepFP is able to achieve a gain of 13.74 %.

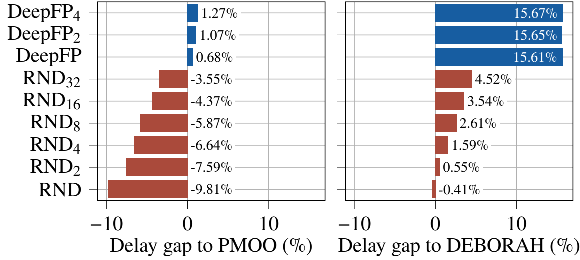


Figure 14: Average delay bound gap of DeepFP to standard PMOO and DEBORAH on the larger networks

Overall, these results illustrate that a simple random choice is not sufficient to improve tightness using flow prolongations. DeepFP is able to accurately choose flow prolongations resulting in a gain in tightness, even on larger networks than the ones it is was trained on.

D. Execution time

To understand the practical applicability of our heuristic, we evaluate in this section its execution time in different settings. We define and measure the execution time per network as the total time taken to analyze all its flows, without including the startup time or the time taken for initializing the network data structures. The execution times were measured on a server with dual AMD EPYC 7542 CPU. The GNN was executed using GPU acceleration with a Nvidia GTX 1080 Ti, while the NC analysis is still executed on CPU. No batching was used, i.e. the GNN analyzes one network at a time.

We first illustrate the average relative execution time of the FP analyses against the non-FP analysis in Figure 15, namely:

$$\frac{\textit{Execution time FP}}{\textit{Execution time non-FP}} \quad (15)$$

This measure helps us understand the cost of using FP. In average, DeepFP with GPU acceleration is approximately an order of magnitude faster than PMOO-FP$_{\text{foi}}$, and almost three orders of magnitude faster than DEBORAH-FP$_{\text{foi}}$. Taking into account the tightness of the method illustrated earlier in Figure 9, those results show that DeepFP is able to achieve a good balance between tightness and computational cost.

DeepFP without GPU acceleration is approximately an order of magnitude faster than DEBORAH-FP$_{\text{foi}}$, making it still an appealing solution despite it's slower execution time. In the case of PMOO-FP$_{\text{foi}}$, DeepFP is actually slower than the exhaustive analysis.

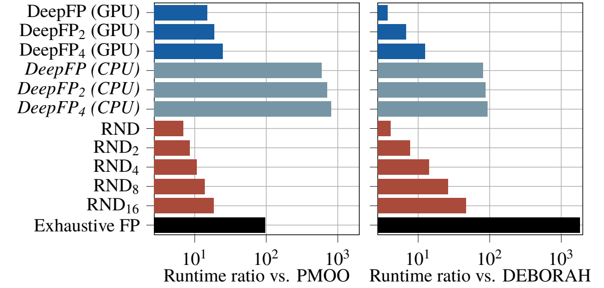


Figure 15: Average relative execution time of different analyses

Second, we evaluate the execution time of the GNN in comparison to the total execution time of the analysis. We use the following measure:

$$\frac{\textit{Execution time GNN}}{\textit{Total execution time}\ (\textit{GNN} + \textit{NC})} \quad (16)$$

Results are presented in Figure 16. When taking advantage of the GPU acceleration, the GNN prediction takes 17.2 % in average of the analysis for PMOO-DeepFP$_{\text{foi}}$, and 2.46 % in average for DEBORAH-DeepFP$_{\text{foi}}$.

Without GPU acceleration, the GNN prediction takes 91.4 % in average of the analysis for PMOO-DeepFP$_{\text{foi}}$, and 64.3 % in average for DEBORAH-DeepFP$_{\text{foi}}$. From Figures 9 and 16, we conclude that DeepFP is mostly attractive in case GPU acceleration is used for the GNN. Despite this drawback, we note that various techniques may be used to speed-up neural network inference on CPU, such as by reducing the size of the GNN, or using mixed-precision floats.

Finally, we evaluate the execution time of the additional enumeration of prolongation combinations used by DeepFP$^+$. As for the GNN part, we use the following measure:

$$\frac{\textit{Execution time Enum.}}{\textit{Total execution time}\ (\textit{Enum.} + \textit{GNN} + \textit{NC})} \quad (17)$$

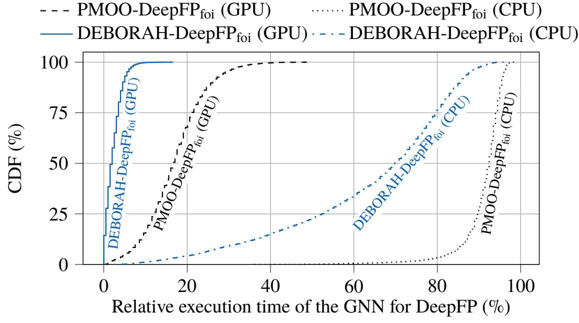


Figure 16: Relative execution time of the GNN for DeepFP

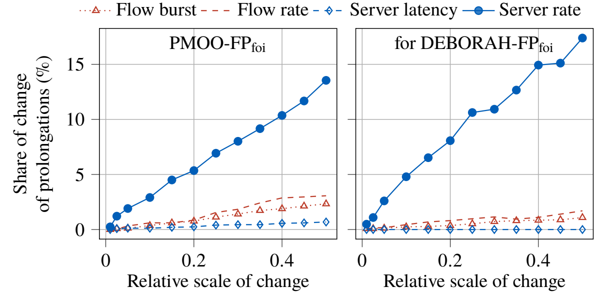


Figure 18: Sensitivity analysis of the NC analyses

Results are presented in Figure 17. In average, the enumeration of prolongation combinations takes 7.22 % of the execution time for PMOO-DeepFP+, and 4.17 % for DEBORAH-DeepFP+. This illustrates that the gains in tightness of DeepFP+ can be achieved at a small computational cost.

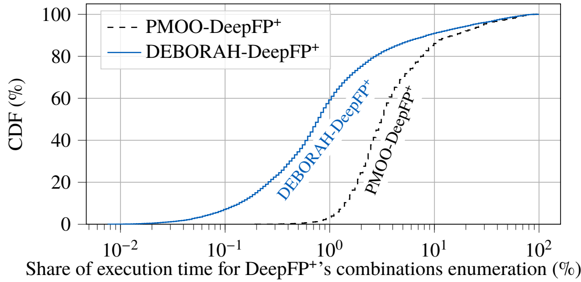


Figure 17: Relative execution time of the GNN for DeepFP

E. Feature importance and sensitivity analysis

We perform here a sensitivity analysis of the choices of flows prolongation of PMOO-FP$_{\mathrm{foi}}$ and DEBORAH-FP$_{\mathrm{foi}}$ to better understand which parameters influence the decision for the best combination. To numerically evaluate this, we randomly modify the curve parameters $p_{original}$ with a relative scale ϵ according to the following uniform distribution:

$$p_{new} \sim \mathcal{U}\left(p_{original}(1-\epsilon), p_{original}(1+\epsilon)\right) \tag{18}$$

We then compare the share of flows where the best combination of flows prolongation have changed due to the random change of curves parameters.

Results are presented in Figure 18. We note that the server's rate has the largest impact on the choice of flows prolongation. Arrival curve parameters also impact also the flows prolongations, but with less magnitude than the service rate. Finally, the service latency has almost no influence on the choice of prolongations, where even large changes of its value result in less than a 1 % change for arbitrary multiplexing, or no changes at all for DEBORAH-FP$_{\mathrm{foi}}$. The service latency, in contrast, is an additive factor in the residual forwarding service computation, making it considerably less impactful.

We use the permutation-based importance measure [56, 57] in order to assess each feature's importance for DeepFP. For each input feature presented in Section V-C, we randomize it by randomly permuting its values in the evaluation set, and assess the impact it has on the relative error of the predictions. We define the feature importance as:

$$\mathit{delay\ bound\ gap}_{\mathrm{foi}}^{Feature} - \mathit{delay\ bound\ gap}_{\mathrm{foi}}^{Baseline} \tag{19}$$

with $\mathit{delay\ bound\ gap}_{\mathrm{foi}}^{Baseline}$ corresponding to the delay bound gap of DeepFP without column permutation.

Results are presented in Figure 19. As expected from the sensitivity analysis, the server rate is the feature having the largest impact on the prediction of the GNN. The other features have almost two orders of magnitude less importance.

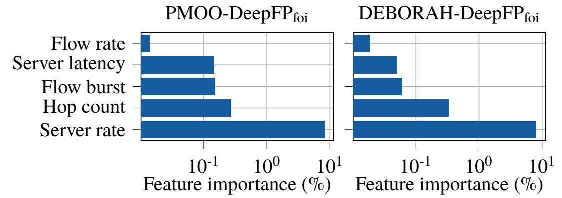


Figure 19: Feature importance of DeepFP

VII. Conclusion

We introduced DeepFP in this paper, an approach for making the NC analysis feature Flow Prolongation scale. FP can be paired with either of the two predominant flow multiplexing assumptions, arbitrary or FIFO, and we show that it is most impactful when bounding the flow of interest's delay (compared to bounding cross-flow arrivals). As each multiplexing assumption's analysis must be trained differently, we devise two analyses: PMOO-FP$_{\mathrm{foi}}$ for arbitrary multiplexing and DEBORAH-FP$_{\mathrm{foi}}$ for FIFO multiplexing. The latter is based on the novel insight that FP can improve the implementation of the PMOO property in the current LUDB FIFO analysis and thus its tool DEBORAH. Our numerical

results show considerably tighter delay bounds of this state-of-the-art algebraic NC FIFO analysis, the average gap to the classic non-FP delay bound rises to 60.75 % – yet at the expense of computational effort. DeepFP predictions solve this problem. We achieve an average accuracy of 69.6 % (PMOO-DeepFP$_{\text{foi}}$) and 60.9 % (DEBORAH-DeepFP$_{\text{foi}}$), resulting in an average relative gap to PMOO-FP$_{\text{foi}}$ of only 1.17 %, and of only 2.57 % to the exhaustive DEBORAH-FP$_{\text{foi}}$ in our first dataset. When scaling to larger networks, where the existing PMOO-FP was known to struggle with computational effort, DeepFP still works. Without considerable loss of prediction accuracy we gain delay bound tightness of 1.06 % compared to standard PMOO, and 13.7 % compared to DEBORAH. In conclusion, we show that FP can considerably tighten NC delay bounds derived for FIFO multiplexing networks and that the proposed GNN-based DeepFP allows to apply it to larger networks.

Appendix

Network Calculus Background [6, 58]

A. Network Calculus System Model

NC models a network as a directed graph of connected queueing locations, the so called server graph. A server offers a resource, in communication networks forwarding of data, and a buffer to queue incoming demand, the data. Data is put into the network by flows. We assume unicast flows with a single source server and a single sink server as well as a fixed route between them. Flows' forwarding demand is characterized by functions cumulatively counting their data,

$$\mathcal{F}_0^+ = \left\{ f : \mathbb{R}^+ \to \mathbb{R}^+ \mid f(0)=0,\ \forall s \leq t\ :\ f(t) \geq f(s) \right\}. \tag{20}$$

Let functions $A(t) \in \mathcal{F}_0^+$ denote a flow's data put into a server and let $A'(t) \in \mathcal{F}_0^+$ be the flow's data put out of, both in the time interval $[0, t)$. We require the input/output relation to preserve causality by $\forall t \in \mathbb{R}^+ :\ A(t) \geq A'(t)$.

NC refines this model to one that uses bounding functions. These univariate functions (called curves) are defined independent of the start of observation, solely based on the duration of the interval of observation. By convention, let curves be in $\mathcal{F}_0$ that simply extends the definition of $\mathcal{F}_0^+$ by $\forall t \leq 0 : f(t)=0$.

Definition 1 (Arrival Curve): Given a flow with input function A, a function $\alpha \in \mathcal{F}_0$ is an arrival curve for A iff

$$\forall\, 0 \leq d \leq t\ :\ A(t) - A(t-d) \leq \alpha(d). \tag{21}$$

Opposite to data arrivals, the forwarding service offered by some system $\mathcal{S}$ is modeled with a lower bounding curve. $\mathcal{S}$ can be a single server as above or – after applying transformations from Appendix B – a combination of multiple servers.

Definition 2 (Service Curve): If the service by system $\mathcal{S}$ for a given input A results in an output A', then $\mathcal{S}$ offers a service curve $\beta \in \mathcal{F}_0$ iff

$$\forall t : A'(t) \geq \inf_{0 \leq d \leq t} \{A(t-d) + \beta(d)\}. \tag{22}$$

Definition 3 (Strict Service Curve): System $\mathcal{S}$ offers a strict service curve β to a flow if, during any busy period of duration d, the output of the flow is at least equal to $\beta(d) \in \mathcal{F}_0$.

In this paper, we restrict the set of curves to affine curves (the only type that can be used with the LUDB analysis). These curves are suitable to model token-bucket shaped data flows $\gamma_{r,b} : \mathbb{R}^+ \to \mathbb{R}^+ \mid \gamma_{r,b}\,(0) = 0,\ \forall_{d>0} \gamma_{r,b}(d) = b + r \cdot d,\ r, b \geq 0$, where b bounds the worst-case burstiness and r the arrival rate. Secondly, rate-latency service can be modeled by affine curves $\beta_{R,T} : \mathbb{R}^+ \to \mathbb{R}^+ \mid \beta_{R,T}\,(d) = \max\{0, R \cdot (d-T)\}$, $T \geq 0$, $R > 0$ where T upper bounds the service latency and R lower bounds the forwarding rate.

B. Algebraic Network Calculus Analysis

The NC analysis aims to derive a bound on the worst-case delay that a specific flow of interest (foi) experiences on its path. Service curves on that path are shared by all flows crossing the respective server yet an arrival curve is only known at the respective flow's source server. To derive the foi's end-to-end delay bound from such a model, the NC analysis relies on (min,plus)-algebraic curve manipulations.

Definition 4 (NC Operations): The (min,plus)-algebraic aggregation, convolution and deconvolution of two functions $f, g \in \mathcal{F}_0$ are defined as

$$\text{aggregation:}\ (f+g)\,(d) = f\,(d) + g\,(d), \tag{23}$$

$$\text{convolution:}\ (f \otimes g)\,(d) = \inf_{0 \leq u \leq d} \{f(d-u) + g(u)\}, \tag{24}$$

$$\text{deconvolution:}\ (f \oslash g)\,(d) = \sup_{u \geq 0} \{f(d+u) - g(u)\}. \tag{25}$$

Aggregation of arrival curves creates a single arrival curve for their multiplex. With convolution, a tandem of servers can be treated as a single system providing a single service curve. Deconvolution allows to compute an arrival curve bound on a flow's (or flow aggregate's) $A'(t)$ after crossing a system. Delay and backlog can be bounded as follows:

Theorem 3 (Performance Bounds): Consider a system $\mathcal{S}$ that offers a service curve β. Assume a flow f with arrival curve α traverses the system. Then we obtain the following performance bounds for f:

$$\text{backlog:}\ \forall t \in \mathbb{R}^+ :\ B\,(t) \leq (\alpha \oslash \beta)\,(0) \tag{26}$$

$$\text{delay:}\ \forall t \in \mathbb{R}^+ :\ D\,(t) \leq \inf\{d \geq 0 \mid (\alpha \oslash \beta)\,(-d) \leq 0\} =: h(\alpha, \beta) \tag{27}$$

When bounding the residual service for a flow of interest (Theorem 1), there are some subtleties to note: the requirement on the service curve β to be strict strongly depends on the assumed multiplexing behavior. Arbitrary multiplexing needs it, FIFO does not [6]. Moreover, the arbitrary multiplexing residual service curve is not strict. In general, arbitrary multiplexing results are bounding those of any other multiplexing assumption. Compared to FIFO, we see that the residual service curves are equal for $\theta = 0$, but for any $\theta > 0$, the FIFO multiplexing can potentially give considerably more residual forwarding service.

ACKNOWLEDGMENTS The authors would like to thank the anonymous shepherd for the feedback and support.

REFERENCES

[1] F. Geyer and G. Carle, "Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches," *IEEE Commun. Mag.*, vol. 54, no. 2, pp. 106–112, 2016.
[2] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht, "Survey on real-time communication via Ethernet in industrial automation environments," in *Proc. of IEEE ETFA*, 2014.
[3] S. Bondorf and F. Geyer, "Generalizing network calculus analysis to derive performance guarantees for multicast flows," in *Proc. of EAI ValueTools*, 2016.
[4] A. Amari and A. Mifdaoui, "Worst-case timing analysis of ring networks with cyclic dependencies using network calculus," in *Proc. of IEEE RTCSA*, 2017.
[5] L. Thomas, J.-Y. Le Boudec, and A. Mifdaoui, "On cyclic dependencies and regulators in time-sensitive networks," in *Proc. of IEEE RTSS*, 2019.
[6] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.
[7] J. B. Schmitt, F. A. Zdarsky, and I. Martinovic, "Improving performance bounds in feed-forward networks by paying multiplexing only once," in *Proc. of GI/ITG MMB*, 2008.
[8] J. B. Schmitt, F. A. Zdarsky, and M. Fidler, "Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch…," in *Proc. of IEEE INFOCOM*, 2008.
[9] S. Bondorf, P. Nikolaus, and J. B. Schmitt, "Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis," *Proc. ACM Meas. Anal. Comput. Syst. (POMACS)*, vol. 1, no. 1, pp. 16:1–16:34, 2017.
[10] M. Boyer, A. Graillat, B. Dupont de Dinechin, and J. Migge, "Bounding the delays of the MPPA network-on-chip with network calculus: Models and benchmarks," *Performance Evaluation*, vol. 143, 2020.
[11] S. Bondorf, "Better bounds by worse assumptions – improving network calculus accuracy by adding pessimism to the network model," in *Proc. of IEEE ICC*, 2017.
[12] P. Nikolaus and J. Schmitt, "Improving delay bounds in the stochastic network calculus by using less stochastic inequalities," in *Proc. of EAI ValueTools*, 2020.
[13] F. Geyer and S. Bondorf, "DeepTMA: Predicting effective contention models for network calculus using graph neural networks," in *Proc. of IEEE INFOCOM*, 2019.
[14] ——, "On the robustness of deep learning-predicted contention models for network calculus," in *Proc. of IEEE ISCC*, 2020.
[15] L. Bisti, L. Lenzini, E. Mingozzi, and G. Stea, "Estimating the worst-case delay in FIFO tandems using network calculus," in *Proc. of ICST ValueTools*, 2008.
[16] ——, "Numerical analysis of worst-case end-to-end delay bounds in FIFO tandem networks," *Real-Time Systems*, vol. 48, no. 5, pp. 527–569, 2012.
[17] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. of ISCAS*, 2000.
[18] A. Bouillard, L. Jouhet, and É. Thierry, "Service curves in network calculus: dos and don'ts," INRIA, Tech. Rep. RR-7094, 2009.
[19] N. Guan and W. Yi, "Finitary real-time calculus: Efficient performance analysis of distributed embedded systems," in *Proc. of IEEE RTSS*, 2013.
[20] Y. Tang, N. Guan, W. Liu, L. T. X. Phan, and W. Yi, "Revisiting GPC and AND connector in real-time calculus," in *Proc. of IEEE RTSS*, 2017.
[21] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan, and W. Yi, "Generalized finitary real-time calculus," in *Proc. of IEEE INFOCOM*, 2017.
[22] Y. Tang, Y. Jiang, X. Jiang, and N. Guan, "Pay-burst-only-once in real-time calculus," in *Proc. of IEEE RTCSA*, 2019.
[23] M. Fidler and V. Sander, "A parameter based admission control for differentiated services networks," *Computer Networks*, vol. 44, no. 4, pp. 463–479, 2004.
[24] A. Bouillard and G. Stea, "Exact worst-case delay for FIFO-multiplexing tandems," in *Proc. of EAI ValueTools*, 2012.
[25] ——, "Exact worst-case delay in FIFO-multiplexing feed-forward networks," *IEEE/ACM Trans. Net.*, vol. 23, no. 5, pp. 1387–1400, 2015.
[26] A. Bouillard, "Trade-off between accuracy and tractability of network calculus in FIFO networks," 2020, arxiv:2010.09263.
[27] S. Bondorf and F. Geyer, "Virtual cross-flow detouring in the deterministic network calculus analysis," in *Proc. of IFIP Networking*, 2020.
[28] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. of IEEE IJCNN*, 2005.
[29] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, 2009.
[30] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018, arxiv:1806.01261.
[31] M. Prates, P. H. Avelar, H. Lemos, L. C. Lamb, and M. Y. Vardi, "Learning to solve NP-complete problems: A graph neural network for decision TSP," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4731–4738.
[32] F. Wang, Z. Cao, L. Tan, and H. Zong, "Survey on learning-based formal methods: Taxonomy, applications and possible future directions," *IEEE Access*, vol. 8, pp. 108 561–108 578, 2020.
[33] K. Rusek and P. Cholda, "Message-passing neural networks learn Little's law," *IEEE Commun. Lett.*, 2018.
[34] F. Geyer, "Performance evaluation of network topologies using graph-based deep learning," in *Proc. of EAI ValueTools*, 2017.
[35] ——, "DeepComNet: Performance evaluation of network topologies using graph-based deep learning," *Performance Evaluation*, 2018.
[36] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN," vol. 38, no. 10, pp. 2260–2270, 2020.
[37] T. Suzuki, Y. Yasuda, R. Nakamura, and H. Ohsaki, "On estimating communication delays using graph convolutional networks with semi-supervised learning," in *Proc. of IEEE ICOIN*, 2020.
[38] T. L. Mai and N. Navet, "Deep learning to predict the feasibility of priority-based Ethernet network configurations," University of Luxembourg, Tech. Rep., 2020.
[39] F. Geyer and S. Bondorf, "Graph-based deep learning for fast and tight network calculus analyses," *IEEE Transactions on Network Science and Engineering*, 2020.
[40] R. L. Cruz, "SCED+: Efficient management of quality of service guarantees," in *Proc. of IEEE INFOCOM*, 1998.
[41] S. Bondorf and J. B. Schmitt, "Should network calculus relocate? an assessment of current algebraic and optimization-based analyses," in *Proc. of QEST*, 2016.
[42] L. Lenzini, E. Mingozzi, and G. Stea, "Delay bounds for FIFO aggregates: A case study," *Comput. Commun.*, vol. 28, no. 3,

pp. 287—299, Feb. 2005.

[43] L. Bisti, L. Lenzini, E. Mingozzi, and G. Stea, “DEBORAH: A tool for worst-case analysis of FIFO tandems,” in *Proc. of ISoLA*, 2010.

[44] S. Bondorf and J. B. Schmitt, “The DiscoDNC v2 – a comprehensive tool for deterministic network calculus,” in *Proc. of EAI ValueTools*, 2014.

[45] ——, “Calculating accurate end-to-end delay bounds – you better know your cross-traffic,” in *Proc. of EAI ValueTools*, 2015.

[46] R. L. Cruz, “A calculus for network delay, part I: Network elements in isolation,” *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 114–131, 1991.

[47] B. Zhou, I. Howenstine, S. Limprapaipong, and L. Cheng, “A survey on network calculus tools for network infrastructure in real-time systems,” *IEEE Access*, vol. 8, 2020.

[48] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” in *Proc. of NIPS*, 2017.

[49] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” in *Proc. of ICLR*, 2016.

[50] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *Proc. of ICLR*, 2018.

[51] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in *Proc. of EMNLP*, 2014.

[52] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. of NeurIPS*, 2019.

[53] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *Proc. of ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[54] A. Guzman-Rivera, D. Batra, and P. Kohli, “Multiple choice learning: Learning to produce multiple structured outputs,” in *Proc. of NIPS*, 2012.

[55] Z. Li, Q. Chen, and V. Koltun, “Combinatorial optimization with graph convolutional networks and guided tree search,” in *Proc. of NIPS*, 2018.

[56] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, 2001.

[57] A. Fisher, C. Rudin, and F. Dominici, “All models are wrong, but many are useful: Learning a variable’s importance by studying an entire class of prediction models simultaneously,” *Journal of Machine Learning Research*, vol. 20, no. 177, pp. 1–81, 2019.

[58] A. Bouillard, M. Boyer, and E. Le Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons, Ltd, 2018.

A.2 Application of graph-based deep learning methods for computer networks

A.2.1 Performance Evaluation of Network Topologies using Graph-Based Deep Learning

This work was published in *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, 2017 [60].

Performance Evaluation of Network Topologies using Graph-Based Deep Learning

Fabien Geyer
Technical University of Munich
fgeyer@net.in.tum.de

ABSTRACT

Understanding the performance of network protocols and communication networks generally relies on expert knowledge and understanding of the different elements of a network, their configuration and the overall architecture and topology. Machine learning is often proposed as a tool to help modeling complex protocols. One drawback of this method is that high-level features are generally used – which require expert knowledge on the network protocols to be chosen, correctly engineered, and measured – and the approaches are generally limited to a given network topology.

In this paper, we propose a methodology to address the challenge of working with machine learning by using lower-level features, namely only a description of the network architecture. Our main contribution is an approach for applying deep learning on network topologies via the use of Graph Gated Neural Networks, a specialized recurrent neural network for graphs. Our approach enables us to make performance predictions based only on a graph-based representation of network topologies. We apply our approach to the task of predicting the throughput of TCP flows. We evaluate three different traffic models: large file transfers, small file transfers, and a combination of small and large file transfers. Numerical results show that our approach is able to learn the throughput performance of TCP flows with good accuracies larger than 90%, even on larger topologies.

CCS CONCEPTS

• **Networks** → **Network performance modeling**; *Transport protocols*; • **Computing methodologies** → **Neural networks**;

KEYWORDS

Network performance evaluation, Graph Neural Network, Deep learning

ACM Reference Format:
Fabien Geyer. 2017. Performance Evaluation of Network Topologies using Graph-Based Deep Learning. In *VALUETOOLS 2017: 11th EAI International Conference on Performance Evaluation Methodologies and Tools, December 5–7, 2017, Venice, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3150928.3150941

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VALUETOOLS 2017, December 5–7, 2017, Venice, Italy
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6346-4/17/12…$15.00
https://doi.org/10.1145/3150928.3150941

1 INTRODUCTION

Understanding network performance is an important task for architecture design and Quality-of-Service in an increasing number of applications. Traffic engineering aims at bringing an answer to this need in order to avoid congestion and optimize network topologies to support an increasing number of applications. Network and traffic models are an important tool in order to predict how a given network architecture will behave. Different techniques have been developed for this purpose, such as mathematical modeling, simulations or measurements. While those techniques are usually accurate, they often require precise measurements of key performance indicators such as round-trip time or loss probability in order to be applied and generate realistic performance predictions. However, limited access to instrumentation of real networks make this measurement acquisition usually difficult.

One growing approach to tackle the challenge of performance modeling has been the use of machine learning. For instance, Tian and Liu applied in [27] the SVR-based (Support Vector Regression) TCP bandwidth prediction application from [19] to improve Quality-of-Service of media streaming over HTTP. Tariq et al. recently proposed WISE in [26], a framework for evaluating architecture changes in communication networks using Causal Bayesian Networks (CBNs). While those techniques and applications have been proven successful, they require high-level features about the studied network protocols and the trained models are often limited to a given network topology.

Our main contribution in this work is an approach for applying deep learning on a low-level graph-based representation of network topologies in order to predict network and protocol performance using only topology information as input. We propose to use Gated Graph Neural Networks (GG-NNs) [17] as a basis for our deep learning architecture. GG-NNs are a recently developed neural network architecture working on graph-structured inputs. The intuition behind our approach is to map network topologies and flows to graphs, and then train GG-NNs on those graphs. This enables us to avoid the task of engineering high-level protocol-specific input features such as round-trip time or drop probability, which usually require expect knowledge on the network protocol which is modeled. Another contribution in this work is the extension of GG-NNs with an alternative memory cell called LSTM (Long Short Term Memory) [13], which shows better performance than the initial architecture from [17].

As a concrete application of our approach, we address the task of performance evaluation of TCP flows, with the goal of predicting the average throughput of each flow in a given topology. We evaluate our approach against three types of traffic models: large file transfers, small file transfers, and a combination of small and large file transfers. As shown in previous studies about TCP, the average

throughput of TCP flows depends on various parameters such as the TCP version, the configuration of the TCP stack, round-trip times or drop probabilities. We show through a numerical evaluation that our approach is able to predict the average throughput of TCP flows without having direct access to those parameters, but only a low-level graph-based representation of the network topology and its flows. The results of our approach are also compared against a simpler recurrent neural network architecture.

This work is structured as follows. In Section 2, we present similar research studies. We describe in Sections 3 and 4 our modeling approach and the neural network architecture used here, with an introduction on Graph Neural Networks and Graph Gated Neural Networks, followed by the application of those concepts to network topologies and flows. We numerically evaluate our approach in Section 5 with the prediction of average flow throughput on three different use-cases. Finally, Section 6 concludes our work.

2 RELATED WORK

On the challenge of predicting the performance of TCP flows, analytical models have been proposed since the late 1990s. Mathis et al., and subsequently Padhye et al., modeled the throughput of a single flow using TCP Reno in [18] and [20] as a function of round-trip time, drop probability and some configuration parameters of TCP. This work was then extended by Cardwell et al. in [5] to take into account the slow-start phase of TCP. While those models address the mathematical modeling of a single flow, the interaction between multiple flows on a given topology is of greater interest for the problem addressed in this paper. Firoiu et al. proposed in [7] to reuse the results from [20] to achieve this. Those analytical models give great insights in the performance of TCP, but they usually suffer from poor applicability in real-world use-cases due to newer versions of TCP, simplifications of the mathematical model, or lack of modeling of non-intuitive behavior of TCP such as ACK compression or TCP Incast. Some later works have partially addressed those shortcomings, such as the work from Velho et al. in [29] and Geyer et al. in [8].

Adjacent to the mathematical modeling of TCP flows, machine learning methods were also applied to this problem, although less frequently than in other domains such as network intrusion detection. Mirza et al. used Support Vector Regression (SVR) in [19] using input features such as transfer duration of files over TCP and active measurements in order to measure queuing latency, loss probability and available bandwidth. Hours et al. used Causal Bayesian Networks (CBNs) in [14] to predict the throughput distribution of TCP flows, using similar features than [19]. Both works showed promising results regarding applicability to real-world use-cases, but are mainly specific to a given network topology or the studied protocol.

Various methods have been proposed for applying machine learning to graphs-based structures, either based on a spectral or spatial approach. Spectral approaches [4, 12] are usually based on the Graph Laplacian, an analogue to the Discrete Fourier Transform, which transforms graph signals to a spectral domain. The main limitation of those approaches is that their requires the input graph samples to be homogeneous.

Spatial approaches do not require a homogeneous graph structure, meaning that they can be applied to a broader range of problems. Gori et al. proposed in [10, 23] the Graph Neural Networks (GNNs) architecture, which propagates hidden representations of nodes to its adjacent nodes until a fixed point is reached. GNNs were applied on different tasks such as object localization, ranking of web pages, document mining, or prediction of graph properties [22]. This neural network architecture was subsequently refined in different works. Li et al. proposed an extension of GNNs in [16] through application of more modern practice of neural networks, namely by using Gated Recurrent Units (GRU) [6]. GG-NNs were applied to basic logical reasoning tasks and program verification in [16]. Relational Graph Convolutional Networks (R-GCNs) were recently proposed by Schlichtkrull et al. in [24], where hidden state information is also propagated across edges of the graph via convolutions, while taking into account the type and direction of an edge.

More general neural network architectures such as the Differentiable Neural Computer from Graves et al. in [11] were also applied to graph-based problems such as shortest path finding.

To the best of our knowledge, this is the first work on applying graph-based neural networks to the performance evaluation of network topologies and network protocols.

3 NEURAL NETWORKS FOR GRAPHS

The main intuition behind our approach is to map network topologies to graphs, with additional nodes for representing flows and additional edges for the path followed by the flows. Those graph representations are then used as input for a neural network architecture able to process general graphs.

In this section, we review the neural network architecture used for this purpose, namely Graph Neural Networks (GNNs) [10, 23] and one of its recent extension, Gated Graph Neural Networks (GG-NNs) [17]. We also introduce notation and concepts that will be used throughout this paper. The transformation between network topology and its graph representation will be detailed later in Section 4.

GNNs and GG-NNs are a general neural network architecture able to process graph structures as input. They are an extension of recursive neural networks which work by assigning hidden states to each node in a graph based on the hidden states of adjacent nodes. For the purpose of this work, our description of GNNs and GG-NNs is limited to undirected graphs. The concepts presented here can also be applied to directed graph, as presented in the original works on GNNs and GG-NNs [10, 17, 23].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Edges can be represented as pairs of nodes, such that $e = (v, v') \in \mathcal{V} \times \mathcal{V}$. The *hidden representation* for node v is denoted by the vector $\mathbf{h}_v \in \mathbb{R}^{\mathcal{D}}$. Nodes may also have features $l_v \in \{1, \ldots, L_{\mathcal{V}}\}$ for each node v, and edges also $l_e \in \{1, \ldots, L_{\mathcal{E}}\}$ for each edge e. Let $\textsc{Nbr}(v)$ denote the set of neighboring nodes of v.

3.1 Graph Neural Networks

In Graph Neural Networks (GNNs), each hidden representation $\mathbf{h}_v$ of a node v is based on the hidden state of its neighboring nodes. The following propagation model is used for expressing this

relationship:

$$\mathbf{h}_v^{(t)} = f^*\left(l_v, l_{\text{NBR}(v)}, \mathbf{h}_{\text{NBR}(v)}^{(t-1)}\right) \quad (1)$$

An example application of Equation (1) is given in Figure 1.

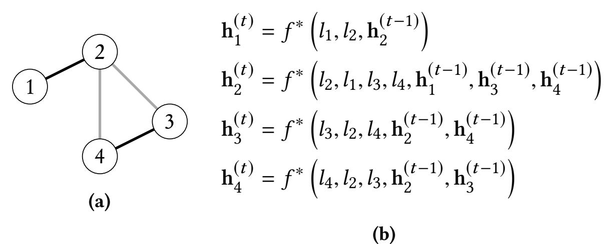


Figure 1: (a) Example graph. Edge colors denote edge types. (b) Application of Equation (1) to the graph.

As a concrete implementation, [23] recommends to decompose $f^*(\cdot)$ as the sum of per-edge terms such that:

$$\mathbf{h}_v^{(t)} = \sum_{v' \in \text{NBR}} f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t-1)}\right) \quad (2)$$

with $f(\cdot)$ a linear function of $\mathbf{h}_v$ or a feed-forward neural network. For example, $f(\cdot)$ can be formulated as a linear function:

$$f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t)}\right) = \mathbf{A}^{(l_v, l_{v'}, l_{(v,v')})}\mathbf{h}_{v'}^{(t-1)} + \mathbf{b}^{(l_v, l_{v'}, l_{(v,v')})} \quad (3)$$

with $\mathbf{A}$ and $\mathbf{b}$ learnable weight and bias parameters. The hidden node representations are propagated throughout the graph until a fixed point is reached. As explained in [23], it implies that $f(\cdot)$ has the property that a fixed point for Equation (2) can be reached.

Once a fixed point $\mathbf{h}_v$ has been reached, a second model is then used to compute the output label o_v for each node $v \in \mathcal{V}$

$$o_v = g\left(\mathbf{h}_v, l_v\right) \quad (4)$$

Practically, $g(\cdot)$ is implemented using a feed-forward neural network. The neural network architecture is differentiable from end-to-end, so that all parameters can be learned using gradient-based optimization.

Learning of the parameters of $f(\cdot)$ and $g(\cdot)$ is done via the Almeida-Pineda algorithm [3, 21] which works by running the propagation of the hidden representation to convergence, and then computing gradients based upon the converged solution.

3.2 Gated Graph Neural Networks

Gated Graph Neural Networks (GG-NNs) [17] are an recent extension of GNNs using more recent neural network techniques, based on Gated Recurrent Units (GRU) [6]. In GG-NNs, each node aggregates the hidden representations it receives from all adjacent nodes, and uses that to update its own hidden representation using a GRU cell. More specifically, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)}\left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \quad (5)$$

$$\mathbf{z}_v^{(t)} = \sigma\left(\mathbf{W}_z a_v^{(t)} + \mathbf{U}_z \mathbf{h}_v^{(t-1)} + \mathbf{b}_z\right) \quad (6)$$

$$\mathbf{r}_v^{(t)} = \sigma\left(\mathbf{W}_r a_v^{(t)} + \mathbf{U}_r \mathbf{h}_v^{(t-1)} + \mathbf{b}_r\right) \quad (7)$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W} a_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^{(t)} \odot \mathbf{h}_v^{(t-1)}\right) + \mathbf{b}\right) \quad (8)$$

$$\mathbf{h}_v^{(t)} = \left(1 - \mathbf{z}_v^{(t)}\right) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{h}_v^{(t)}} \quad (9)$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\{\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}\}$ and $\{\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}\}$ are learnable weights matrices, and $\{\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}\}$ are learnable biases vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is a matrix determining how nodes in the graph $\mathcal{G}$ communicate with each other, as illustrated in Figure 2.

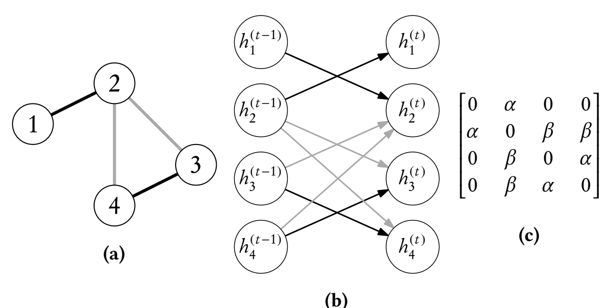


Figure 2: (a) Example graph. Edge colors denote edge types. (b) One time-step unrolling of Equations (5) to (9). (c) Matrix A corresponding to the graph. Parameters α and β encode the edge type.

Equation (5) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for Graph Neural Networks in Equations (1) and (2). Equations (6) to (9) correspond to the mathematical formulation of a GRU cell, with Equation (6) representing the GRU reset gate vector, Equation (7) the GRU update gate vector, and Equation (9) the GRU output vector. The initial hidden representation $h_v^{(0)}$ is based on the node's feature vector l_v, padded with zeros according to the dimensions of the hidden representation.

The output vector $\mathbf{o}_v$ for each node v is computed as in Equation (4) using a feed-forward neural network. The overall architecture of the GG-NN is summarized in Figure 3.

Learning of the weight matrices and bias vectors is performed using back-propagation through time in order to compute gradients, namely using standard gradient-based optimization algorithms such as RMSProp [28] or Adam [15].

3.3 GG-LSTM-NN: Extension of Graph Gated Neural Networks with LSTM

We propose in this section a new class of Graph Gated Neural Networks called GG-LSTM-NN based on the Long Short-Term Memory (LSTM) cell [13]. This neural network architecture is a variant of the

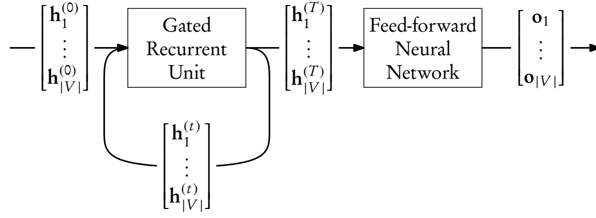


Figure 3: Representation of a Gated Graph Neural Network.

Graph Gated Neural Network architecture where the GRU memory cell is replaced with a LSTM cell. The overall architecture of the GG-LSTM-NN is similar to the GG-NN illustrated in Figure 3.

Similarly to Equations (5) to (9), the propagation of the hidden representations among neighboring nodes for one time-step in a GG-LSTM-NN is formulated as:

$$\mathbf{a}_v^{(t)} = A_{(v)} \left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \quad (10)$$

$$\mathbf{i}_v^{(t)} = \sigma\left(\mathbf{W}_i a_v^{(t)} + \mathbf{U}_i \mathbf{h}_v^{(t-1)} + \mathbf{b}_i\right) \quad (11)$$

$$\mathbf{f}_v^{(t)} = \sigma\left(\mathbf{W}_f a_v^{(t)} + \mathbf{U}_f \mathbf{h}_v^{(t-1)} + \mathbf{b}_f\right) \quad (12)$$

$$\mathbf{o}_v^{(t)} = \sigma\left(\mathbf{W}_o a_v^{(t)} + \mathbf{U}_o \mathbf{h}_v^{(t-1)} + \mathbf{b}_o\right) \quad (13)$$

$$\mathbf{g}_v^{(t)} = \tanh\left(\mathbf{W}_g a_v^{(t)} + \mathbf{U}_g \mathbf{h}_v^{(t-1)} + \mathbf{b}_g\right) \quad (14)$$

$$\mathbf{c}_v^{(t)} = \mathbf{c}_v^{(t-1)} \odot \mathbf{f}_v^{(t)} + \mathbf{g}_v^{(t)} \odot \mathbf{i}_v^{(t)} \quad (15)$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{c}_v^{(t)}\right) \odot \mathbf{o}_v^{(t)} \quad (16)$$

with $\{\mathbf{W}_i, \ldots\}$ and $\{\mathbf{U}_i, \ldots\}$ learnable weight matrices, and $\{\mathbf{b}_i, \ldots\}$ learnable bias vectors.

Equation (10) is the propagation of the hidden representations among neighbors, as in Equation (5). Equations (11) to (13) correspond respectively to the *input*, *forget* and *output* gates of the LSTM cell. Equation (14) is a *candidate* hidden representation, with an initial value $\mathbf{c}_v^{(0)}$ set to zero. Equation (15) is the internal memory of the LSTM cell.

Our motivation for proposing this new neural network architecture is motivated by better numerical performance than the GRU-based GG-NN presented in Section 3.2, as shown in the numerical evaluation in Section 5.

4 APPLICATION TO PERFORMANCE EVALUATIONS OF NETWORKS

We describe in this section the application of the deep learning architectures presented earlier to the performance evaluation of network topologies and network protocols. In other words, our goal is to represent network topologies and their flows as graphs which can be passed as input to a GG-NN. Compared to other works on the application of machine learning to performance evaluation, the main contribution is that this graph representation is a low-level input feature. This means that specific high-level features of the studied network protocol are not required and the trained machine learning algorithm is not restricted to a specific topology.

4.1 Input features definition

The main intuition behind the input feature modeling for the GG-NN is to use the queuing network as input graph $\mathcal{G}$, with additional nodes representing the flows in this network. An illustration of this queuing network is given in Figure 5, which is the queuing representation of the example network illustrated in Figure 4 with one switch or router interconnecting three PCs with three flows. Note that we illustrate on Figure 5 only the forward path of the flows. Figure 5 may also be extended to include the queues taken by the acknowledgement packets used by the flows if necessary, such as TCP ACK packets for example.

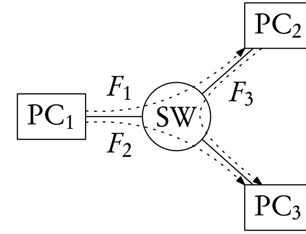


Figure 4: Example network topology with 3 flows.

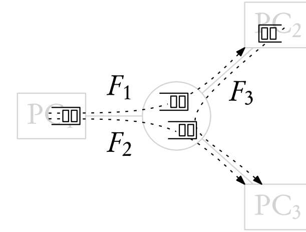


Figure 5: Associated queuing network of Figure 4 (here with only the forward path of each flow).

Regarding the constructed graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the nodes $\mathcal{V}$ correspond to the queues traversed by the flows in the network topology as well as specific nodes representing the flows. For the node features l_v, a vector encoding the node type (*i.e.* if a node represents a flow or a queue) with one-hot encoding is used. Namely l_v is a vector with two values, with $[1, 0]^T$ is for queue nodes, and $[0, 1]^T$ for flow nodes. Note that for simplification purpose, we assume here that every PCs and switches or routers to have the same behavior and all links in the topology to have the same capacity and latency. Additional features for distinguishing between different behaviors or node types may be used in case different configurations, types or link capacities are used. An example of such node-specific feature is given later in Sections 5.3 and 5.4.

Edges connect the queues which are used by the flows according to the physical topology of the network. In order to encode flow routing in the graph, edges between the nodes representing flows and their traversed queues are used. Figure 6 is an example of such graph modeling applied to the topology presented in Figure 4. A labeling of the edge type may be used in order to distinguish

between queues representing the forward path of flows and the path used for acknowledgement packets, as illustrated in Figure 2.

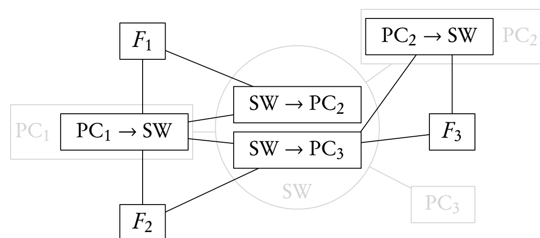


Figure 6: Graph representing the queuing network from Figure 5.

We note that the graph and feature representations described here are independent of any studied network protocols.

4.2 Task-specific modeling

For the scope of this work, we focus on the challenge of evaluating the performance of TCP flows, namely predicting their average throughput. As shown in previous works on TCP Reno such as [18], the average throughput of a TCP flow can be modeled as:

$$\frac{C}{RTT\sqrt{p}} \tag{17}$$

with RTT the round-trip time and p the probability of packet loss on the flow's path, and C a constant value depending on a configuration of the TCP stack. In the case of small file transfers, additional parameters such as file size distributions are also important.

This is an interesting problem since TCP adapts its throughput according to the perceived level of congestion in the network or other factors depending on the version of TCP which is used. Since the flows also contribute themselves to the overall congestion in the network, the sharing of bandwidth at a given bottleneck is not trivial to predict.

Based on this task description, the output vector $\mathbf{o}_v$ of node v will then be the average throughput of the TCP flow for nodes representing flows. Practical experimentations showed that using discretized values across N bins provided better accuracy than using continuous values for modeling the throughputs. Hence the regression task essentially becomes a classification task.

The neural network is then trained against a log softmax cross entropy loss function, as usually done in classification tasks:

$$\mathcal{L}_v = -y_v + \log \sum_{i=1}^{N} e^{\mathbf{o}_{v:i}} \tag{18}$$

where y_v corresponds to the index of the binned average throughput value, and $\mathbf{o}_{v:i}$ to the i-th element of the output vector $\mathbf{o}_v$.

5 NUMERICAL EVALUATION

We present in this section a numerical evaluation of the concepts presented in Sections 3 and 4. We focus here on the evaluation of Ethernet networks with 100 Mbit/s links. The evaluated topologies and flows are randomly generated as follows using [9]. A random number of Ethernet switches is first selected using a uniform distribution and connected in according to a daisy chain as illustrated in Figure 7. A random number of nodes is then generated using a uniform distribution and connected to a randomly selected Ethernet switch. For each node, a TCP flow is generated with a randomly selected destination among the other nodes.

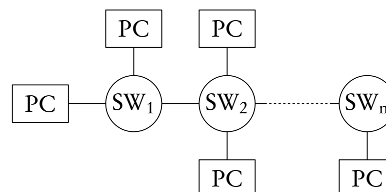


Figure 7: Daisy chain topology used for the numerical evaluation.

In order to build our datasets for learning, each random topology is evaluated using the *ns-2* simulator [1] until the steady-state of the flows throughput is reached. The defaults parameters of *ns-2* for the TCP stack are used, meaning that TCP Reno is used as a congestion control algorithm. The results of the simulations is used as a basis for the learning process of the neural network.

We evaluate our approach against three different traffic use-cases, namely:

(1) Infinite flows, where clients always have data to send, with results presented in Section 5.2;
(2) Finite flows, where clients follow an ON/OFF loop behavior where a random amount of data is sent, followed by a random idle time, with results presented in Section 5.3;
(3) A combination of the two previous traffic models, with results presented in Section 5.4.

5.1 Implementation

The GG-NN architectures presented in Sections 3.2 and 3.3 was implemented using Tensorflow [2]. The recurrent part of the GG-NN and GG-LSTM-NN were respectively implemented according to Equations (5) to (9) and Equations (10) to (16). The function $g(\cdot)$ in Equation (4) was implemented using a feed-forward neural network with two dense layers. Additional dropout layers [25] between each time-step of the GG-NN were added in order to avoid over-fitting.

For each studied use-case, the model was trained multiple times using the parameters listed in Table 1 and different seeds for the random number generators. Randomization of the node indexes was also performed for each mini-batch. The neural network producing the best result was then selected for the numerical results presented in the rest of this section.

As a comparison basis, we also evaluated a simple version of Graph Neural Network from section 3.1, using a simple Recurrent Neural Network (RNN) architecture similar. The hidden node representation is driven by:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)} \left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \tag{19}$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\mathbf{h}_v^{(t-1)} + \mathbf{b}\right) \tag{20}$$

Parameter	Value
Learning algorithm	RMSProp [28]
Size of hidden representation	64
Learning rate	10^{-4}
Mini-batch size	32
Training iterations	20 000

Table 1: Parameters used for the training phase of the neural networks.

with **W** and **U** learnable weight matrices, and **b** a learnable bias vector.

Each set of simulations was split into training and test datasets. The training dataset was used for training the neural network, while the test dataset was used for evaluating the prediction performances of the neural network.

5.2 Evaluation on infinite TCP flows

In this first use-case, we assume an infinite traffic model for TCP flows. This model illustrates the case of large file transfers over TCP. Multiple sets of simulations were generated, with different parameters regarding the size of the network, namely the maximum number of switches used for the line topology presented in Figure 7 and the maximum number of flows. Statistics about the dataset with the largest topologies are given in Table 2.

Property	Mean	Min.	Max.	Std. dev.
Number of flows	16.38	2	30	8.21
Number of queues	35.54	4	66	16.75
Number of edges	250.47	20	596	133.10

Table 2: Parameters of the dataset with the largest network topologies.

Numerical results of the average accuracy of the predictions of the trained neural network are presented on Figure 8. As mentioned in Section 4.2, the output vector corresponds to a discretized value of the average throughput of the evaluated TCP flows. The accuracy is hence defined as the correct prediction from the neural network of the bins associated to the average throughputs.

We notice in Figure 8 that the GG-NN described in Section 3.2 – labeled *GG-GRU-NN* in the plot – is able to reach accuracies higher than 85 %, even on topologies with a larger number of flows and number of hops. On small topologies, the neural network is able to reach accuracies higher than 95 %. We notice that the average prediction accuracy decreases slightly with the complexity of the network, namely according to the number of hops traversed by the TCP flows and overall number of flows in the network.

The results of the GG-LSTM-NN architecture, our proposition for a modification of the GG-NN architecture, performs better than the original GG-NN architecture, with overall accuracies higher than 90 %. Finally, as a comparison, the RNN architecture from Equation (20) is only able to reach accuracies higher than 90 %, even on the smaller topologies.

Those numerical results motivates our choice of the GG-LSTM-NN architecture over the original GG-NN and RNN architectures for our application.

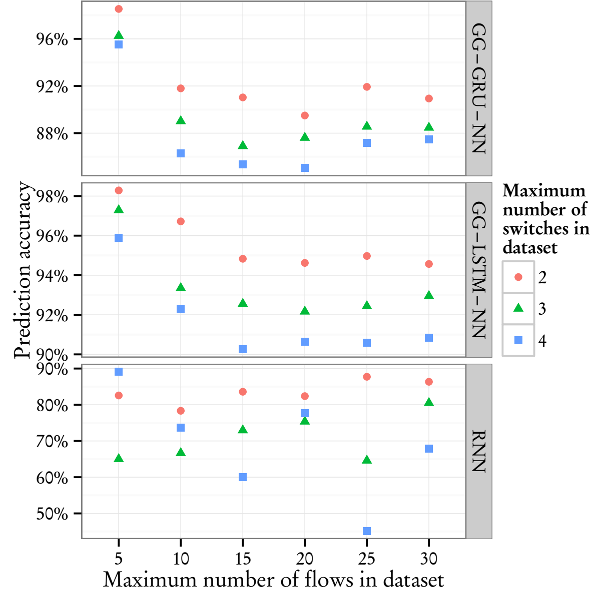


Figure 8: Average prediction accuracy on topologies with infinite TCP flows. Each data point correspond to another dataset where the neural networks were trained and evaluated.

5.3 Evaluation on finite TCP flows

In this second use-case, we assume an ON/OFF traffic model for TCP flows, where clients repeatedly send a random amount of data followed by a random idle period. It represents the case of small file transfers over TCP. This traffic model is illustrated in Figure 9 where three flows are sharing the same link. We restrict here the line topology to a maximum of two switches.

An exponential distribution is used for file sizes, where each flow is randomly assigned a different mean value between 1 MB and 5 MB for the file size distribution. Similarly, an exponential distribution is used for idle periods, with a mean value of 1 s for all flows. The feature vector l_v is extended here to take into account the mean of the file size distribution, using one-hot encoding.

Numerical results are presented in Figure 10. Despite less accurate predictions compared to Figure 8, we notice that the neural networks are still able to learn the bandwidth sharing of the ON/OFF flows and use the file size distributions. As in the previous results, the GG-LSTM-NN architecture outperforms the GRU-based GG-NN architecture.

In order to illustrate the impact of file size distribution on the prediction accuracy, we also trained the neural network without this information. Results are presented in Figure 11. As expected, the prediction accuracy decreases, showing that the network was able to use the file size distribution previously in Figure 10.

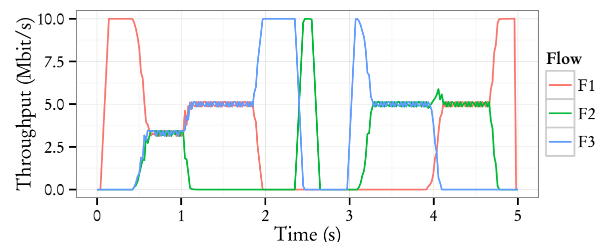


Figure 9: Illustration of the bandwidth sharing of three ON/OFF TCP flows using the same bottleneck. Each data point correspond to another dataset where the neural networks were trained and evaluated.

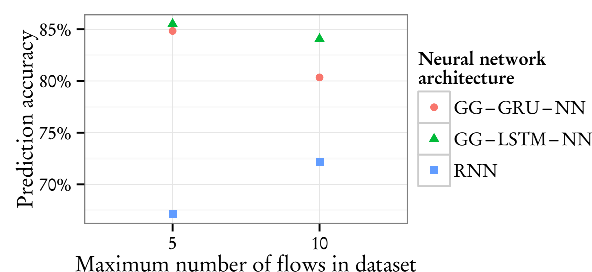


Figure 10: Average prediction accuracy on topologies with finite ON/OFF TCP flows. Each data point correspond to another dataset where the neural networks were trained and evaluated.

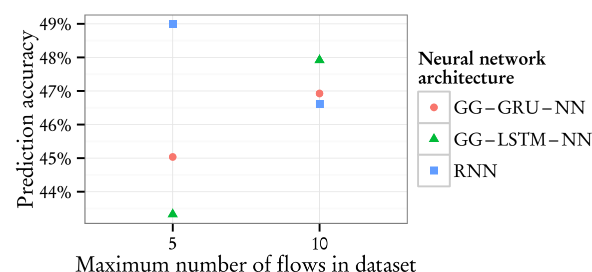


Figure 11: Average prediction accuracy on topologies with finite ON/OFF TCP flows, where the neural network is trained without information on the file size distributions.

5.4 Evaluation on combined infinite and ON/OFF TCP flows

In this third use-case, we assume a combination of both previous traffic models on the same network, where some flows are infinite and some flows are finite. This combined traffic model is often referred as *"mice and elephants"* in the literature. The same parameters as in Section 5.3 were used for the file size distributions and idle time distributions. Topologies were generated such that $1/6^{\text{th}}$ of the flows were infinite flows, and the rest finite ON/OFF flows as presented in Figure 9. We also restrict here the line topology to a maximum of two switches.

Numerical results are presented in Figure 12. We notice here similar results than in Figure 10. The neural networks are able to predict the average throughputs, although with less accuracy compared to the two previous use-cases.

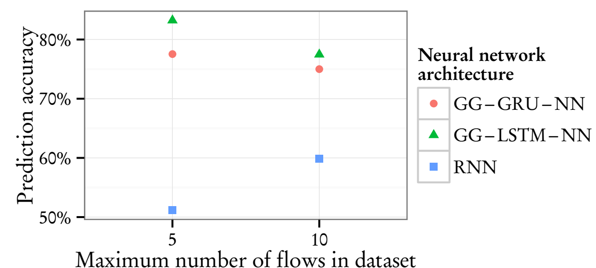


Figure 12: Average prediction accuracy on topologies with a combination of finite and infinite TCP flows.

5.5 Interpreting GG-NNs

An important subject when working with neural network is the interpretability of the learned weights. We propose here to visualize Equation (5) as t increases, namely visualize how the hidden representation of a node evolves at different time-steps. This is illustrated in Figure 13 on a sample network with 4 flows and 10 queues.

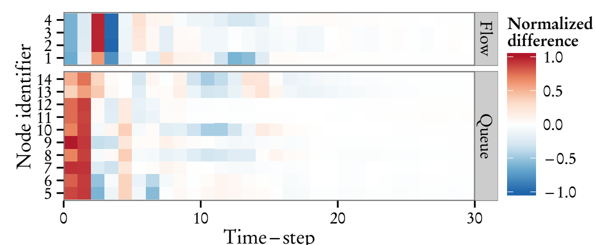


Figure 13: Visualization of the hidden representations' propagation. The color of each cell corresponds to $\sum_{\mathcal{D}} \mathbf{h}_v^{(t)} - \mathbf{h}_v^{(t-1)}$ for each node v in the graph.

We observe that a fixed point for Equation (5) is indeed reached since there is almost no difference between $\mathbf{h}_v^{(t)}$ and $\mathbf{h}_v^{(t-1)}$ in the last time-steps for all nodes. It is interesting to notice that the fixed point for each queue is reached at different time-steps, meaning that some queues (eg. node 13 or 14) have a larger impact on flow performance than other queues (eg. node 5 or 6).

6 CONCLUSION

We presented in this paper a novel approach for the performance evaluation of network topologies and flows by using graph-based deep learning. Our approach is based on the use of a modified Gated

Graph Neural Networks called GG-LSTM-NN and a low-level graph-based representation of queues and flows in network topologies. Compared to other approaches using machine learning for performance evaluation of computer networks, the trained model is not specific to a given topology and high-level input features requiring more advanced knowledge on the studied protocol are not required.

We applied our approach to the performance evaluation of TCP flows with the task of predicting the average throughput for each flow. This is an interesting task since the throughput of TCP flows is dependent on the network architecture and network conditions (*i.e.* congestion and delays). Different traffic models for the flows were evaluated: large file transfers, small file transfers, and a combination of large and small file transfers. We showed via a numerical evaluation that our approach is able to reach good accuracies, even on large network topologies with multiple hops. We compared the chosen neural network architecture against a simpler recurrent neural network architecture, motivating our choice for GG-NNs. Finally we also visualized the internal working of the neural network in order to give some insights on which queues have an influence on protocol performances.

Since the network topology is directly taken as input of the neural network, applications such as network planning and architecture optimization may benefit from the method developed in this paper. As our approach is not specific to the performance evaluation of TCP flows, future work may include evaluations and extensions of our approach to other congestion control algorithms, performance measure such as latency or other network protocols.

Acknowledgments This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16KIS0538 (DecADe).

REFERENCES

[1] 2017. ns-2, Network Simulator (ver. 2.35). (2017). Retrieved July 28, 2017 from https://www.isi.edu/nsnam/ns/

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[3] Luis B. Almeida. 1990. Artificial Neural Networks. IEEE Press, Piscataway, NJ, USA, Chapter A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment, 102–111.

[4] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR'2014)*.

[5] Neal Cardwell, Stefan Savage, and Thomas Anderson. 2000. Modeling TCP Latency. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, Vol. 3. IEEE, 1742–1751. https://doi.org/10.1109/INFCOM.2000.832574

[6] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. (June 2014). arXiv:1406.1078

[7] Victor Firoiu, Ikjun Yeom, and Xiaohui Zhang. 2001. A Framework for Practical Performance Evaluation and Traffic Engineering in IP Networks. In *Proceedings of the IEEE International Conference on Telecommunications*.

[8] Fabien Geyer, Stefan Schneele, and Georg Carle. 2013. Practical Performance Evaluation of Ethernet Networks with Flow-Level Network Modeling. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2013)*. 253–262. https://doi.org/10.4108/icst.valuetools.2013.254367

[9] Fabien Geyer, Stefan Schneele, and Georg Carle. 2014. PETFEN: A Performance Evaluation Tool for Flow-Level Network Modeling of Ethernet Networks. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2014)*. https://doi.org/10.4108/icst.valuetools.2014.258166

[10] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A New Model for Learning in Graph Domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks (IJCNN'05)*, Vol. 2. IEEE, 729–734. https://doi.org/10.1109/IJCNN.2005.1555942

[11] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (Oct. 2016), 471–476. https://doi.org/10.1038/nature20101

[12] Mikael Henaff, Joan Bruna, and Yann LeCun. 2015. Deep Convolutional Networks on Graph-Structured Data. (June 2015). arXiv:1506.05163

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[14] Hadrien Hours, Ernst W. Biersack, and Patrick Loiseau. 2016. A Causal Approach to the Study of TCP Performance. *ACM Trans. Intel. Syst. Tech.* 7, 2 (Jan. 2016), 25:1–25:25. https://doi.org/10.1145/2770878

[15] Diederik Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'2015)*.

[16] Cheng Li, Xiaoxiao Guo, and Qiaozhu Mei. 2016. DeepGraph: Graph Structure Predicts Network Growth. (Oct. 2016). arXiv:1610.06251

[17] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'2016)*.

[18] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Comput. Commun. Rev.* 27, 3 (June 1997), 67–82. https://doi.org/10.1145/263932.264023

[19] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. 2010. A Machine Learning Approach to TCP Throughput Prediction. *IEEE/ACM Trans. Netw.* 18, 4 (Aug. 2010), 1026–1039. https://doi.org/10.1109/TNET.2009.2037812

[20] Jitendra Padhye, Victor Firoiu, Don F. Towsley, and James F. Kurose. 2000. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Trans. Netw.* 8, 2 (April 2000), 133–145. https://doi.org/10.1109/90.842137

[21] Fernando J. Pineda. 1987. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.* 59 (Nov. 1987), 2229–2232. Issue 19. https://doi.org/10.1103/PhysRevLett.59.2229

[22] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. Computational Capabilities of Graph Neural Networks. *IEEE Trans. Neural Netw.* 20, 1 (Jan. 2009), 81–102. https://doi.org/10.1109/TNN.2008.2005141

[23] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Netw.* 20, 1 (Jan. 2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[24] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. (March 2017). arXiv:1703.06103

[25] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (Jan. 2014), 1929–1958.

[26] Mukarram Bin Tariq, Kaushik Bhandankar, Vytautas Valancius, Amgad Zeitoun, Nick Feamster, and Mostafa Ammar. 2013. Answering "What-If" Deployment and Configuration Questions With WISE: Techniques and Deployment Experience. *IEEE/ACM Trans. Netw.* 21, 1 (Feb. 2013), 1–13. https://doi.org/10.1109/TNET.2012.2230448

[27] Guibin Tian and Yong Liu. 2012. Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, 109–120. https://doi.org/10.1145/2413176.2413190

[28] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning* 4, 2 (2012), 26–31.

[29] Pedro Velho, Lucas M. Schnorr, Henri Casanova, and Arnaud Legrand. 2011. *Flow-level network models: have we reached the limits?* Technical Report 7821. INRIA.

A.2.2 DeepComNet: Performance Evaluation of Network Topologies using Graph-Based Deep Learning

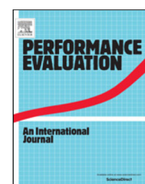
This work was published in *Performance Evaluation*, 2019 [61].

Performance Evaluation 130 (2019) 1–16

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva



DeepComNet: Performance evaluation of network topologies using graph-based deep learning

Fabien Geyer

Technical University of Munich, Boltzmannstr. 3, 85748 Garching b. München, Germany

A R T I C L E I N F O

Article history:
Available online 28 December 2018

Keywords:
Network performance evaluation
Graph neural network
Deep learning

A B S T R A C T

Modeling the performance of network protocols and communication networks generally relies on expert knowledge and understanding of the different elements of a network, their configuration, and the overall architecture and topology. Machine learning is often proposed as a tool to help modeling such complex systems. One drawback of this method is that high-level features are generally used – which require full understanding of the network protocols to be chosen, correctly engineered, and measured – and the approaches are generally limited to a given network topology.

In this article, we present *DeepComNet*, an approach to address the challenges of working with machine learning by using lower-level features, namely only a description of the network architecture. Our main contribution is a method for applying deep learning on network topologies via the use of Graph Gated Neural Networks, a specialized recurrent neural network for graphs. Our approach enables us to make performance predictions based only on a graph-based representation of network topologies. To evaluate the potential of DeepComNet, we apply our approach to the tasks of predicting the throughput of TCP flows and the end-to-end latencies of UDP flows. In both cases, the same base model is used. Numerical results show that our approach is able to learn and predict performance properties of TCP and UDP flows with a median absolute relative error smaller than 1%, outperforming related methods from the literature by one order of magnitude.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Understanding network performance is an important task for architecture design and Quality-of-Service in an increasing number of applications. Traffic engineering aims at bringing an answer to this need in order to provide better services, avoid congestion, and optimize network topologies to support an increasing number of applications. Models for network and traffic are an important tool in order to predict how a given network architecture will behave. Different techniques have been proposed, such as mathematical modeling, simulations or measurements in real networks. While these techniques can achieve accurate results, they often require precise measurements of key performance indicators such as round-trip time or loss probability in order to be applied and generate realistic performance predictions. However, limited access to instrumentation of real networks makes this measurement acquisition usually difficult.

An approach to tackle the challenges of performance modeling is a more data-driven way, where measurements are used in parallel with machine learning to produce realistic performance predictions. For instance, Tian and Liu [1] applied the SVR-based (Support Vector Regression) TCP bandwidth prediction application from [2] to improve Quality-of-Service of media streaming over HTTP. Tariq et al. [3] recently proposed WISE, a framework for evaluating architecture changes

E-mail address: fgeyer@net.in.tum.de.

https://doi.org/10.1016/j.peva.2018.12.003
0166-5316/© 2018 Elsevier B.V. All rights reserved.

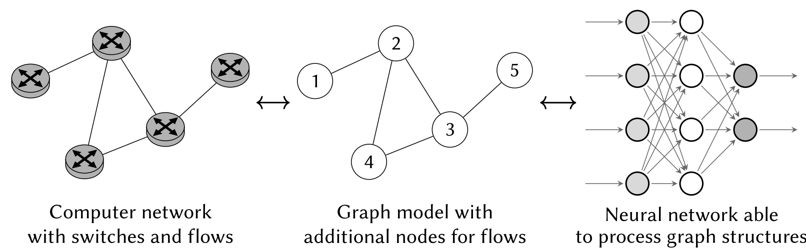


Fig. 1. Illustration of the two different steps of our approach *DeepComNet*.

in communication networks using Causal Bayesian Networks (CBNs). While these techniques and applications have been proven successful, high-level features about the studied network protocols are required, and the trained models are often limited to a given network topology.

Our main contribution in this work is *DeepComNet*, an approach combining formal modeling of network topologies and flows traversing them as graphs with deep learning methods tailored for training and inference on graph structures. This approach is illustrated in Fig. 1 with its two main steps: we first model computer networks as graph structures, and then process those graphs using neural networks. We propose to use Gated Graph Neural Networks (GGNNs) [4] as a basis for our deep learning architecture, a recent extension of Graph Neural Networks (GNNs) [5,6]. By directly training the underlying machine learning algorithm on the network structure, we avoid the task of engineering high-level protocol-specific input features such as round-trip time or drop probability, which usually require expert knowledge on the network protocol which is modeled. We demonstrate in this paper this modeling as graph is independent of a given network architecture or network protocol, providing a wide range of applicability compared to other methods.

As concrete applications of our approach, we address the two following tasks. First, we are interested in the performance evaluation of TCP flows, with the goal of predicting the average throughput of each flow in a given topology. Secondly, we look at the performance evaluation of UDP flows and the prediction of end-to-end latencies (average and quantile value). We use the same model for both tasks, highlighting the general applicability of our approach and potential for application to other use-cases for performance evaluation of network topologies and protocols.

We show through a numerical evaluation that our approach is able to predict the performance of TCP and UDP flows by only using this low-level graph-based representation of the network topology and its flows. Overall, the predictions of bandwidths and end-to-end latencies have a median absolute relative error smaller than 1%. The results for the TCP dataset are also compared against two approaches using high-level input features (round-trip time and loss probability), namely Support Vector Regression (SVR) as used in [1,2] and a standard feed-forward neural network. Compared to both approaches, our method achieves better accuracy, with a median relative error one order of magnitude lower compared to SVR.

We also illustrate the scalability of our approach in terms of execution time by numerically evaluating our implementation of GNN. Our measurements illustrate that our implementation scales quadratically with the number of edges, and linearly with the number of nodes. We show that our approach is able to process graphs with up to 1000 nodes and 150 000 edges with a prediction time below 200 us. Finally we illustrate a method for visualizing the learned model in order to apply it to more general questions such as root-cause analysis or bottleneck identification.

This work is structured as follows. In Section 2, we present similar research studies. We describe in Sections 3 and 4 our modeling approach and the neural network architecture used here, with an introduction to Graph Neural Networks and Gated Graph Neural Networks, followed by the application of those concepts to network topologies and flows. We numerically evaluate our approach in Section 5 with the prediction of performance of TCP and UDP flows, compare against two other machine learning approaches, and evaluate our approach in terms of scalability. Section 6 gives some insights on interpreting the learned weights of the neural network. Finally, Section 7 concludes our work.

2. Related work

As shown in a recent survey by Fadlullah et al. [7], machine learning has been used for computer networks for various applications such as traffic classification, flow prediction, and mobility prediction. It has also been applied in the area of performance evaluation of communication networks. Mirza et al. [2] used Support Vector Regression (SVR) using input features such as transfer duration of files over TCP and active measurements in order to measure queuing latency, loss probability and available bandwidth. Tariq et al. [3] proposed WISE to study application performance using Causal Bayesian Networks (CBNs) in order to answer "what-if" questions. Hours et al. [8] also used CBNs to predict the throughput distribution of TCP flows, using similar features as in [2]. While these works proved to deliver accurate results, they are essentially based on high-level engineered features dependent on the studied protocol and those models are not easily transferable to other protocols. In case of TCP flows, these high level features usually include round-trip time and packet loss – following well known mathematical models such as [9] – which also often require active measurement. By having a simple process for abstracting the network topology as graph without such high-level feature, the approach we propose in this article does not

require such expert knowledge or hand engineered features and is better suited to be used on multiple use-cases or multiple network protocols. Our approach is also able to reason about performance evaluation at network level and not at individual node level.

Various methods have been proposed for applying machine learning to graphs-based structures, either based on a spectral or spatial approach. Spectral approaches [10,11] are usually based on the Graph Laplacian, an analogous method to the Discrete Fourier Transform, which transforms graph signals to a spectral domain. The main limitation of these approaches is that the input graph samples are required to be homogeneous. Concretely this means that only a fixed subset of network types may be analyzed and multiple machine learning models may be required in case multiple network types are required.

Spatial approaches do not require a homogeneous graph structure, meaning that they can be applied to a broader range of problems. Gori et al. proposed in [5,6] the Graph Neural Networks (GNNs) architecture, which propagates hidden representations of nodes to its adjacent nodes until a fixed point is reached. GNNs were applied on different tasks such as object localization, ranking of web pages, document mining, or prediction of graph properties [12]. This neural network architecture was subsequently refined in different works. Li et al. [4] proposed Gated Graph Neural Networks (GGNNs), an extension of GNNs with application of more modern practice of neural networks, namely by using Gated Recurrent Units (GRU) [13]. GNNs were evaluated against basic logical reasoning tasks and program verification in [4] and shown to be as-good-as or better than previous state of the art methods. Related approaches were proposed by Kipf and Welling [14] with Graph Convolutional Networks (GCN) and later extended by Schlichtkrull et al. [15] with Relational Graph Convolutional Networks (R-GCNs). GCN also use hidden node state information propagated across edges of the graph via convolutions, while taking into account the type and direction of an edge. Finally, Battaglia et al. [16] recently introduced the graph networks (GN) framework with the goal of providing a unified formalization of many concepts applied in GNNs and extensions of GNNs.

Those methods have been used in a variety of applications related to the study of data which can be modeled as graphs. Grover and Leskovec [17] applied it to link prediction and classification in social networks, protein interactions and natural language processing. Marcheggiani and Titov [18] studied semantic role labeling in natural language processing. Gilmer et al. [19] used GNNs for the prediction of chemical properties of molecules. Allamanis et al. [20] performed source code analysis by modeling source code and interaction between program variables as graphs, in order to predict variable naming. Selsam et al. [21] modeled the interaction between variables in boolean satisfiability problems (SAT) as graphs, with the goal of predicting their satisfiability. In all those applications, the results of graph-based deep learning were similar to or better than previously existing machine learning approaches.

On the challenge of predicting the performance of TCP flows, analytical models have been proposed since the late 1990s. Mathis et al. [22], and subsequently Padhye et al. [9], modeled the throughput of a single flow using TCP Reno as a function of round-trip time, drop probability and some configuration parameters of TCP. This work was then extended by Cardwell et al. [23] to take into account the slow-start phase of TCP. While these models address the mathematical modeling of a single flow, the interaction between multiple flows on a given topology is of greater interest for the problem addressed in this paper. Firoiu et al. [24] proposed to reuse the results from [9] to analyze complete topologies. Those analytical models give great insights in the performance of TCP, but they usually suffer from poor applicability to real-world use-cases due to newer versions of TCP, simplifications of the mathematical model, or lack of modeling of non-intuitive behavior of TCP such as ACK compression or TCP Incast. Some later works have partially addressed those shortcomings, such as the work from Velho et al. [25] and Geyer et al. [26].

This article is an extension of our previous work [27], which applied GGNNs to the performance evaluation of TCP flows. In this article, we propose more advanced GGNNs architectures, provide an extended numerical evaluation with the performance of both TCP and UDP flows, compare against two other machine learning approaches using high-level features, and provide a discussion regarding scalability.

3. Neural networks for graph analysis

The main intuition behind our approach is to represent network topologies as graphs with additional nodes for modeling flows and additional edges for the paths followed by the flows. These graph representations are then used as input for a neural network architecture able to process general graphs. The transformation between a given network topology and its graph representation will be detailed later in Section 4.

In this section, we review the neural network architecture used for training neural networks on graphs, namely Graph Neural Networks (GNNs) [5,6] and one of its recent extension, Gated Graph Neural Networks (GGNNs) [4]. We also introduce notation and concepts that will be used throughout this article. Alternate approaches for applying neural networks to graphs were presented in Section 2.

GNNs and GGNNs are a general neural network architecture able to process graph structures as input. They are an extension of recursive neural networks which work by assigning hidden states to each node in a graph based on the hidden states of adjacent nodes. For the purpose of this work, our description of GNNs and GGNNs is limited to undirected graphs. The concepts presented here can also be applied to directed graph, as presented in the original works on GNNs and GGNNs [4–6].

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Edges can be represented as pairs of nodes, such that $e = (v, v') \in \mathcal{V} \times \mathcal{V}$. The *hidden representation* for node v is denoted by the vector $\mathbf{h}_v \in \mathbb{R}^{\mathcal{H}}$. Nodes may also have features l_v for each node v, and edges also $l_e = l_{(v,v')}$ for each edge $e = (v, v')$. Let $\textsc{Nbr}(v)$ denote the set of neighboring nodes of v.

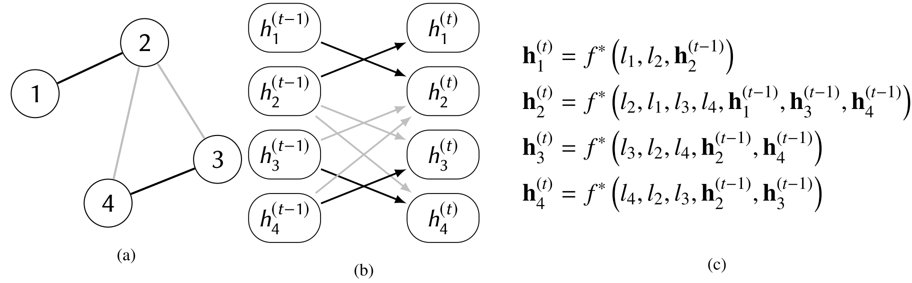


Fig. 2. (a) Example graph. Edge colors denote edge types. (b) One step of the propagation of hidden node representation. (c) Application of Eq. (1) to the graph.

3.1. Graph neural networks

In Graph Neural Networks (GNNs), each hidden representation $\mathbf{h}_v$ of a node v is based on the hidden state of its neighboring nodes $\textsc{Nbr}(v)$. The following propagation model is used for expressing this relationship:

$$\mathbf{h}_v^{(t)} = f^*\left(l_v, l_{\textsc{Nbr}(v)}, \mathbf{h}_{\textsc{Nbr}(v)}^{(t-1)}\right) \tag{1}$$

An illustrative application of Eq. (1) is given in Fig. 2.

As a concrete implementation, [6] recommends to decompose $f^*(\cdot)$ as the sum of per-edge terms such that:

$$\mathbf{h}_v^{(t)} = \sum_{v' \in \textsc{Nbr}} f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t-1)}\right) \tag{2}$$

with $f(\cdot)$ a linear function of $\mathbf{h}_v$ or a feed-forward neural network. For example, $f(\cdot)$ can be formulated as a linear function:

$$f\left(l_v, l_{v'}, l_{(v,v')}, \mathbf{h}_{v'}^{(t)}\right) = \mathbf{W}^{(l_v, l_{v'}, l_{(v,v')})}\mathbf{h}_{v'}^{(t-1)} + \mathbf{b}^{(l_v, l_{v'}, l_{(v,v')})} \tag{3}$$

with $\mathbf{W}$ and $\mathbf{b}$ learnable weight and bias parameters. The hidden node representations are propagated throughout the graph until a fixed point is reached. As explained in [6], it implies that $f(\cdot)$ has the property that a fixed point for Eq. (2) can be reached.

Once a fixed point $\mathbf{h}_v$ has been reached, a second model is then used to compute the output vector $\mathbf{o_v}$ for each node $v \in \mathcal{V}$ such that:

$$\mathbf{o_v} = g\left(\mathbf{h}_v, l_v\right) \tag{4}$$

Practically, $g(\cdot)$ is implemented using a feed-forward neural network [6]. The neural network architecture is differentiable from end-to-end, so that all parameters can be learned using standard techniques for neural networks based on gradient-based optimization.

Due to the required fixed-point iterations, training of the parameters of $f(\cdot)$ and $g(\cdot)$ of the GNN is done via the Almeida–Pineda algorithm [28,29] which works by running the propagation of the hidden representation to convergence, and then computing gradients based upon the converged solution.

3.2. Gated graph neural networks

Li et al. [4] recently introduced Gated Graph Neural Networks (GGNNs) as an extension of GNNs using more recent neural network techniques, based on Gated Recurrent Units (GRU) [13]. GRUs are special types of neural network blocks with an internal memory. Such blocks are generally used to process time-dependent inputs such as audio or written words, which can be mathematically simplified as:

$$output^{(t)} = function\left(input^{(t)}, output^{(t-1)}\right) \tag{5}$$

In GGNNs, each node aggregates the hidden representations it receives from all adjacent nodes as in Eq. (2), and uses that to update its own hidden representation using a GRU cell. More specifically, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)}\left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \tag{6}$$

$$\mathbf{z}_v^{(t)} = \sigma\left(\mathbf{W}_z a_v^{(t)} + \mathbf{U}_z \mathbf{h}_v^{(t-1)} + \mathbf{b}_z\right) \tag{7}$$

$$\mathbf{r}_v^{(t)} = \sigma\left(\mathbf{W}_r a_v^{(t)} + \mathbf{U}_r \mathbf{h}_v^{(t-1)} + \mathbf{b}_r\right) \tag{8}$$

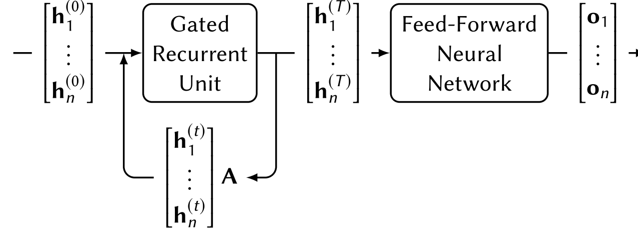


Fig. 3. Representation of a Gated Graph Neural Network.

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}a_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^{(t)} \odot \mathbf{h}_v^{(t-1)}\right) + \mathbf{b}\right) \quad (9)$$

$$\mathbf{h}_v^{(t)} = \left(1 - \mathbf{z}_v^{(t)}\right) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{h}_v^{(t)}} \quad (10)$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\{\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}\}$ and $\{\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}\}$ are learnable weights matrices, and $\{\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}\}$ are learnable biases vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining how nodes in the graph $\mathcal{G}$ communicate with each other.

Eq. (6) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for Graph Neural Networks in Eqs. (1) and (2). Eqs. (7)–(10) correspond to the mathematical formulation of a GRU cell, with Eq. (7) representing the GRU reset gate vector, Eq. (8) the GRU update gate vector, and Eq. (10) the GRU output vector as described in Eq. (5). The initial hidden representation $\mathbf{h}_v^{(0)}$ is based on the node's feature vector l_v, padded with zeros according to the dimensions of the hidden representation.

The output vector $\mathbf{o}_v$ for each node v is computed as in Eq. (4) using a feed-forward neural network. The overall architecture of the GGNN is illustrated and summarized in Fig. 3.

While with GNNs the node propagation loop is performed until a fixed-point is reached, a fixed number of iterations is used in GGNNs. This process enables the use of traditional gradient-based training methods used for feed-forward neural networks such as RMSProp [30]. Since hidden representations are propagated iteratively to node neighbors at each iteration, the number of iterations necessary for achieving good accuracy depends mainly on the path length between relevant nodes in the analyzed graph. In the example in Fig. 2, if the output vector of node 1 depends on the input features of node 4, at least two iterations of the propagation loop are required, since the hidden representation of node 4 is first propagated to node 2 in the first iteration, before reaching node 1 in the second iteration.

While the number of iterations is highly dependent on application, we propose here two simple heuristics for choosing the number of iterations. The first heuristic is to use the diameter d of the studied graph, namely the greatest distance between any pair of nodes, which ensures that each node receives at least once the propagated hidden representation of all other nodes in the graph. The second heuristic is to use $2 \times d$, which ensures that the hidden representations between any pairs of nodes in the studied graph could be propagated in both directions. Numerical evaluations in Section 5.7 illustrate the influence of the number of iteration on the accuracy.

3.3. GGNN-LSTM: Extension of gated graph neural networks with LSTM

In our approach we use an extension of Gated Graph Neural Networks – called here GGNN-LSTM – based on the Long Short-Term Memory (LSTM) cell [31]. This neural network architecture is based on the Gated Graph Neural Network architecture where the GRU memory cell is replaced with a LSTM cell. The overall architecture of the GGNN-LSTM is similar to the GGNN illustrated in Fig. 3.

Similarly to Eqs. (6)–(10), the propagation of the hidden representations among neighboring nodes for one time-step in a GGNN-LSTM is formulated as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)}\left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \quad (11)$$

$$\mathbf{i}_v^{(t)} = \sigma\left(\mathbf{W}_i\mathbf{a}_v^{(t)} + \mathbf{U}_i\mathbf{h}_v^{(t-1)} + \mathbf{b}_i\right) \quad (12)$$

$$\mathbf{f}_v^{(t)} = \sigma\left(\mathbf{W}_f\mathbf{a}_v^{(t)} + \mathbf{U}_f\mathbf{h}_v^{(t-1)} + \mathbf{b}_f\right) \quad (13)$$

$$\mathbf{o}_v^{(t)} = \sigma\left(\mathbf{W}_o\mathbf{a}_v^{(t)} + \mathbf{U}_o\mathbf{h}_v^{(t-1)} + \mathbf{b}_o\right) \quad (14)$$

$$\mathbf{g}_v^{(t)} = \tanh\left(\mathbf{W}_g\mathbf{a}_v^{(t)} + \mathbf{U}_g\mathbf{h}_v^{(t-1)} + \mathbf{b}_g\right) \quad (15)$$

$$\mathbf{c}_v^{(t)} = \mathbf{c}_v^{(t-1)} \odot \mathbf{f}_v^{(t)} + \mathbf{g}_v^{(t)} \odot \mathbf{i}_v^{(t)} \quad (16)$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{c}_v^{(t)}\right) \odot \mathbf{o}_v^{(t)} \quad (17)$$

with $\{\mathbf{W}_i, \ldots\}$ and $\{\mathbf{U}_i, \ldots\}$ learnable weight matrices, and $\{\mathbf{b}_i, \ldots\}$ learnable bias vectors.

Eq. (11) is the propagation of the hidden representations among neighbors, as in Eq. (6). Eqs. (12)–(14) correspond respectively to the *input*, *forget* and *output* gates of the LSTM cell. Eq. (15) is a *candidate* hidden representation, with an initial value $\mathbf{c}_v^{(0)}$ set to zero. Eq. (16) is the internal memory of the LSTM cell.

Our rationale for using this alternate neural network architecture is motivated by better accuracy than the GGNN based on GRU presented in Section 3.2, as shown in the numerical evaluation in Section 5 for the TCP task.

3.4. Stacked gated graph neural networks

Since the neural network architectures introduced earlier are based on a special variant of recurrent neural networks, advances made for standard RNNs may also be applied here. For this purpose, we make use of stacked RNNs as proposed by Pascanu et al. [32], and illustrated in the following equation:

$$\mathbf{h}^{(t;l)} = r\left(A, \mathbf{h}^{(t-1;l)}, \mathbf{h}^{(t;l-1)}\right) \tag{18}$$

where $\mathbf{h}^{(t;l)}$ corresponds to the hidden representation at timestep t and layer l, and r a function describing the recurrent unit, i.e. Eqs. (6)–(10) for the GRU or Eqs. (12)–(14) for the LSTM unit.

3.4.1. Edge attention

A recent advance in neural networks has been the concept of *attention*, which provides the ability to a neural network to focus on a subset of its inputs. This mechanism has been used in a variety of applications such as computer vision or natural language processing (e.g. [33]). For the scope of GNNs, we introduce here so-called *edge attention*, namely we wish to give the ability to each node to focus only on a subset of its neighborhood. Formally, let $a_{(v,u)}^{(t)} \in [0, 1]$ be the attention between node v and u. Eq. (2) is then extended as:

$$\mathbf{h}_v^{(t)} = \sum_{u \in \textsc{Nbr}(v)} a_{(v,u)}^{(t)} \cdot f^* \left(\mathbf{h}_u^{(t-1)}\right) \tag{19}$$

$$\mathbf{a}_{(v,u)}^{(t)} = f_A\left(\mathbf{h}_v^{(t-1)}, \mathbf{h}_u^{(t-1)}\right) \tag{20}$$

We decompose $f_A(\cdot)$ as two feed-forward neural networks and use an element-wise matrix multiplication to compute a modified adjacency matrix.

To make the computation of $a_{(v,u)}^{(t)}$ for all edges more efficient, we use the modified adjacency matrix $\widetilde{A}^{(t)}$ defined as:

$$\widetilde{A}^{(t)} = A \odot \mathit{softmax}\left(f_{A_1}\left(\mathbf{H}^{(t)}\right)^T \odot f_{A_2}\left(\mathbf{H}^{(t)}\right)\right) \tag{21}$$

with $f_{A_1}(\cdot)$ and $f_{A_2}(\cdot)$ feed-forward neural networks, and *softmax* the normalized exponential function such that:

$$\mathit{softmax}(\mathbf{X})_j = \frac{e^{X_j}}{\sum_i e^{X_i}} \tag{22}$$

4. Application to performance evaluations of networks

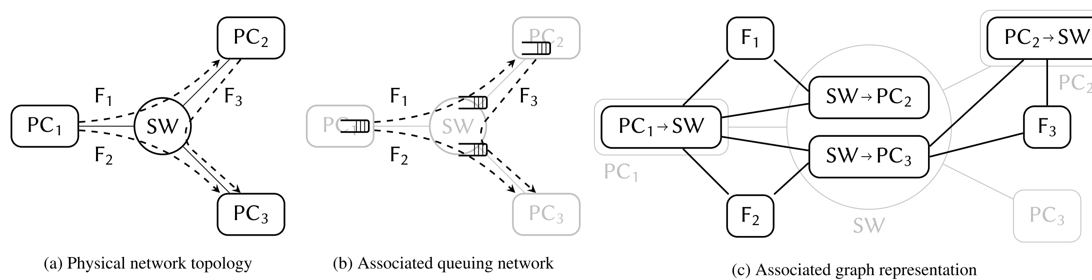
We describe in this section the application of the deep learning architectures presented earlier to the performance evaluation of network topologies and network protocols. In other words, our goal is to represent network topologies and the flows traversing them as graphs which can be passed as inputs to a GGNN. Compared to other works on the application of machine learning to performance evaluation, the main contribution is that this graph representation is a low-level input feature. This means that specific high-level features of the studied network protocol are not required and the trained machine learning algorithm is not restricted to a specific topology.

The main intuition behind the input feature modeling for the GGNN is to use the network of queues as input graph $\mathcal{G}$, with additional nodes representing the flows in this network. An illustration of this network of queues is given in Fig. 4b, which is the representation of the different queues in the example network depicted in Fig. 4a. Note that we represent on Fig. 4b only the forward path of the flows. Fig. 4b may also be extended to include the queues taken by the acknowledgment packets used by the flows if necessary, such as TCP ACK packets for example as explained later in Section 5.2.

We formally define here the transformation between an input computer network and the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which will be analyzed by the GGNN. Fig. 4c is an application of this transformation applied to the network presented in Fig. 4a.

We focus in this article on the study of simple Ethernet networks, where computers are connected via simple store-and-forward switches and exchanging data via flows. We assume that each flow in the network is unidirectional, with a single source and a single destination. Each flow f is represented as a node v_f in the graph $\mathcal{G}$. As illustrated in Section 5.2, bidirectional flows such as TCP flows may still be analyzed by using two unidirectional flows, modeled in $\mathcal{G}$ by two nodes connected by an edge.

We assume here that the routing in the studied network is static, namely that the path taken by a flow is fixed. Each queue traversed by a flow in the network is represented as a node v_q in the graph $\mathcal{G}$. For simplifications purpose we only



(a) Physical network topology (b) Associated queuing network (c) Associated graph representation

Fig. 4. Graph encoding of an example network topology with three flows.

consider the queues present in the network interfaces and not other queues which may be present in operating systems. We model network interfaces using a simple drop-tail queue. In case of Ethernet switches, multiple nodes are used since multiple interfaces are present. Queues are connected by edges according to the physical topology of the studied network. Let Q_f be the set of queues traversed by flow f. In order to encode the path taken by a flow, edges connect the node representing the flow v_f with the nodes representing the traversed queues $\{v_q | q \in Q_f\}$.

Each node v in the graph $\mathcal{G}$ is characterized by its input features l_v, encoding parameters relevant to the studied network protocol. In the simple case illustrated here, we only encode the node type – *i.e.* if a node represents a flow or a queue – using one-hot encoding. Namely l_v is a vector with two values, with $[1, 0]^T$ is for queue nodes, and $[0, 1]^T$ for flow nodes.

Each node v in the graph $\mathcal{G}$ is also characterized by its output features o_v corresponding to the values which are predicted. In this article, we study in Section 5 the case where the output vector o_{v_f} of each flow f corresponds to a bandwidth or an end-to-end latency. Since we do not make prediction on queues here, their corresponding nodes have no output vector. This is implemented by using a masked loss function for training the neural network, which only take into account the output vectors of flows.

Note that for simplification purpose, we limited this section – and description of transformation between network and graph – to the case where all nodes in the network have the same behavior and that all links in the topology have the same capacity and latency. Additional features for distinguishing between different behaviors or node types (e.g. link capacity, different vendors, operating system) may easily be added in case differences between nodes are present and relevant to the prediction, such as different configurations, types or link capacities. In this case, l_v is extended with additional values representing those differences and encoded using standard practices for feature encoding in machine learning. Example of such scenario where each node has a unique attribute is given later in Section 5 where flow rates and packet sizes are additionally encoded in l_v.

We note that this network transformation process may easily be extended to more complex architectures and more refined models of queues, such as having network interfaces using multiple queues and a packet scheduler, or modeling more precisely network switches using additional queues depending on the internal architecture of switches. Since standard packet processing pipelines can easily be represented as graphs, additional nodes and edges representing more complex pipelines may be added in the graph.

Compared to more traditional approaches as in [2,3,8], we note that the graph and feature representations described here are independent of any studied network protocols or specific metrics (e.g. bandwidth, latency, etc.).

5. Numerical evaluation

In this section, we evaluate Graph Neural Networks in the context of our approach described in Sections 3 and 4 on two representative use-cases: prediction of the average steady-state bandwidth of TCP flows and prediction of end-to-end latencies of UDP flows.

5.1. Implementation

For both use-cases, the same implementation, parameters and learning algorithm for the neural network were used. The GGNN architectures presented in Sections 3.2 and 3.3 were implemented using Tensorflow [34] and trained using a Nvidia GeForce GTX 1080Ti GPU. The recurrent part of the GGNN and GGNN-LSTM were respectively implemented according to Eqs. (6)–(10) and Eqs. (11)–(17). The function $g(\cdot)$ in Eq. (4) was implemented using a feed-forward neural network with two dense layers. Additional dropout layers [35] were added according to standard practices for recurrent neural networks [36] in order to avoid over-fitting.

For each studied use-case, the model was trained multiple times using the parameters listed in Table 1. The learning rate used for training the models was optimized following standard practices based on Bayesian optimization. The learned weights producing the best result is then selected for the numerical results presented in the rest of this section. All GNN models evaluated here were trained for the same number of iterations.

Table 1
Parameters used for the training phase of the neural networks.

Parameter	Value
Size of hidden representation $\mathbf{h}_v$	96
Number of loops unrolling	12
Training algorithm	RMSProp [30]
Mini-batch size	64
Number of training iterations	20 000

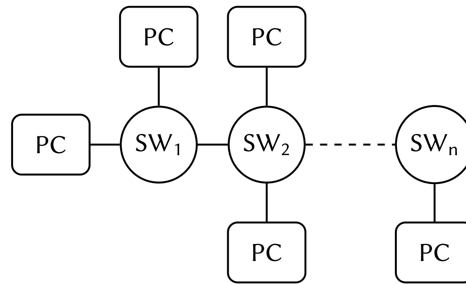


Fig. 5. Daisy chain topology used for the numerical evaluation.

The following Graph Neural Network architectures were evaluated:

- **GGNN-GRU:** A Gated Graph Neural Network using a GRU memory unit as presented in Section 3.2,
- **GGNN-LSTM:** A Gated Graph Neural Network using a LSTM memory unit as presented in Section 3.3,
- **GNN-RNN:** A simplified implementation of Graph Neural Network as described in Section 3.1 using a simple Recurrent Neural Network (RNN) architecture, where the hidden node representation is computed as:

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{(v)} \left[\mathbf{h}_1^{(t-1)} \cdots \mathbf{h}_{|V|}^{(t-1)}\right]^T + \mathbf{b}_a \tag{23}$$

$$\mathbf{h}_v^{(t)} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\mathbf{h}_v^{(t-1)} + \mathbf{b}\right) \tag{24}$$

  with $\mathbf{W}$ and $\mathbf{U}$ learnable weight matrices, and $\mathbf{b}$ a learnable bias vector.
- **GNN-RNN^2** and **GNN-RNN^3** as stacked versions of GNN-RNN following Section 3.4, with respectively 2 and 3 stacks, as indicating by the exponent in the architecture label.

In order to compare our approach with other machine learning approaches, we evaluate also the following additional models:

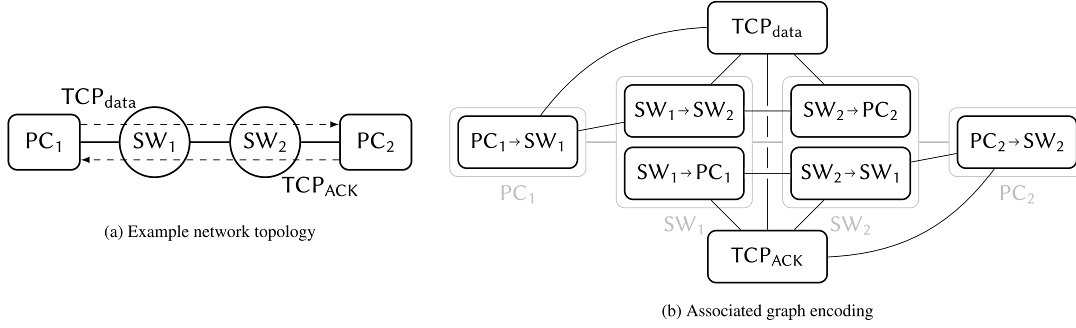
- **SVR:** Support Vector Regression for the TCP bandwidth predictions using measured round-trip time and loss probability as input features. This model is comparable to one proposed by Mirza et al. [2]. We note that the use of such model requires active measurement in the network. The implementation of SVR from `scikit-learn` [37] was used for the evaluation.
- **FFNN:** A feed-forward neural network with three dense layers for the TCP bandwidth predictions using the same features as the SVR model.

5.2. Use-case: prediction of average TCP flow bandwidths

In this first use-case, we evaluate the capabilities of our approach at predicting the steady-state bandwidth of TCP flows sharing different bottlenecks.

For the generation of the topologies, a random number of Ethernet switches is first selected using the discrete uniform distribution $\mathcal{U}(1, 6)$ and connected according to a daisy chain as illustrated in Fig. 5. A random number of nodes is then generated using the discrete uniform distribution $\mathcal{U}(2, 32)$ and connected to a randomly selected Ethernet switch. For each node, a TCP flow is generated with a randomly selected destination among the other nodes. The default parameters of *ns-2* for the TCP stack are used, meaning that TCP Reno is used as a congestion control algorithm. The results of the simulations are used as a basis for the learning process of the neural network.

Since the performance of TCP flows is highly dependent on the congestion experienced by TCP acknowledgments, we extend our approach from Section 4 to also include TCP ACKs in the input graph representation. Fig. 6 represents the graph encoding which has been used for this task on a simple topology. We follow the method explained in Section 4, where each queue in the network is represented by a node. Each TCP flows is encoded as two nodes in the graph: one node for representing the *data sub-flow* and one node for the *acknowledgment sub-flow*, both connected by an edge. Each node i in the graph has a feature vector encoding node type as a one-hot vector:



(a) Example network topology

(b) Associated graph encoding

Fig. 6. Graph encoding used for the TCP prediction task.

- $[1, 0, 0]^T$ for nodes representing queues,
- $[0, 1, 0]^T$ for TCP data nodes,
- $[0, 0, 1]^T$ for TCP acknowledgment nodes.

The output vector $\mathbf{y}_i$ of each node representing a TCP data sub-flow is a single value corresponding to the normalized steady-state bandwidth. The output vector of the other nodes is masked in the loss function.

As shown in previous studies about TCP [9,24], the average throughput of TCP flows depends on various parameters such as the TCP congestion control algorithm, the configuration of the TCP stack, round-trip times or drop probabilities, as well as the interaction between flows. The graph encoding used here offers a more low-level representation.

The neural network is trained against a mean-square loss function such that the following function is minimized:

$$\frac{1}{|flows|} \sum_{i \in flows} (\mathbf{y}_i - \mathbf{o}_i)^2 \tag{25}$$

5.3. Use-case: prediction of UDP flows end-to-end latencies

In this second use-case, we evaluate the capabilities of our approach at predicting the end-to-end latencies of UDP flows sharing different bottlenecks. We follow the same approach as for the previous use-case, namely multiple random topologies with UDP flows are generated and evaluated using simulations. The same daisy-chain topology presented in Fig. 5 is used here.

Each UDP flow i generates unidirectional traffic from a given pair of source S_i and destination D_i, with constant rate r_i, using packets of random size taken from a uniform distribution from 10 to m_i Bytes. The parameters S_i, D_i, r_i and m_i are randomly generated for each UDP flow. For this use-case we focus on the prediction of the average and 95% quantile of the end-to-end delay for each flow in a given topology.

Regarding the graph encoding, we follow the approach illustrated in Section 4 with Fig. 4 since flows are unidirectional. As for the TCP use-case, the feature vector of each node i encodes the node type as a one-hot value and the traffic parameters:

- $[1, 0, \widetilde{r}_i, \widetilde{m}_i]^T$ for nodes representing flows, with $\widetilde{r}_i$ and $\widetilde{m}_i$ the normalized versions of r_i and m_i
- $[0, 1, 0, 0]^T$ for nodes representing queues

The output vector $\mathbf{y}_i$ of each node representing a UDP flow corresponds to the normalized end-to-end latency. As for the TCP use-case, we use Eq. (25) as loss function, where the output vector of the non-flow nodes are masked.

5.4. Prediction accuracy

In order to evaluate the accuracy of the different models presented here, we evaluate the predictions using the relative absolute error and residual metrics:

$$\text{Relative absolute error:} \quad \frac{|\mathbf{y}_i - \mathbf{o}_i|}{\mathbf{y}_i} \tag{26}$$

$$\text{Residual:} \quad \mathbf{y}_i - \mathbf{o}_i \tag{27}$$

with $\mathbf{y}_i$ and $\mathbf{o}_i$ respectively measurements from the simulations, and predicted values.

Fig. 7 illustrates the distributions of relative error of the different neural network architectures for the datasets previously described.

We first focus our interpretation of Fig. 7 on the TCP use-case with machine learning models using high-level features (round-trip time and loss probability), namely SVR and FFNN. The median relative absolute error of 11.6% for the SVR model

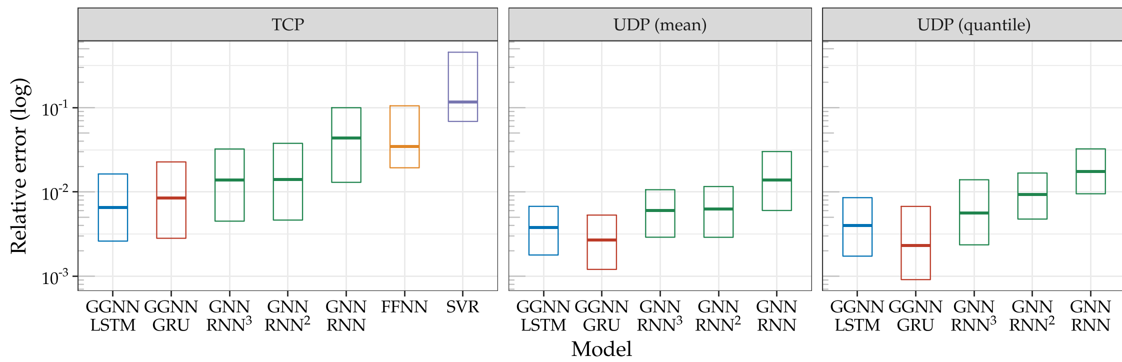


Fig. 7. Comparison of the evaluated machine learning methods according to the relative error. Bars represent respectively the 25, 50 and 75 percentile values.

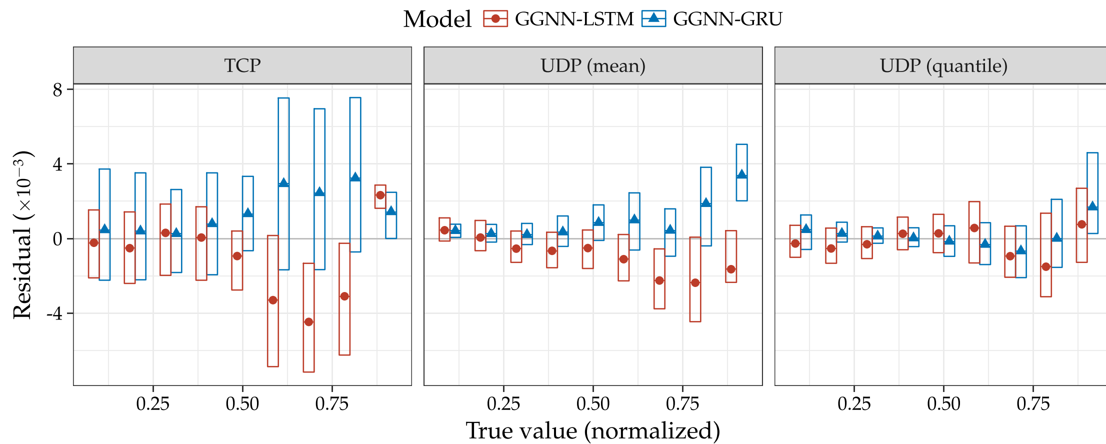


Fig. 8. Comparison of real value against the residual ($\mathbf{o}_i - \mathbf{y}_i$) for the GGNN-GRU and GGNN-LSTM architectures. Values are normalized according to the studied dataset. Bars represent respectively the 25, 50 and 75 percentile values.

is in line with the results from [2], which reported a 10% median relative error. The feed-forward neural network provides better results, with a median relative error of 3.5%. Those values will be used as a baseline for comparison purposes.

Regarding the GGNN models, all architectures evaluated here are able to predict the TCP bandwidths with a median relative error below 1%, outperforming the values from the SVR by one order of magnitude, and also outperforming the feed-forward neural network using high-level input features. This highlights our main motivation for using GGNNs.

Regarding UDP latencies, we are also able to reach a median relative error below 1%. The GGNN-LSTM architectures provide better results on the TCP use-case, while the GRU-based ones are more suited to the UDP use-case. We note that using stacked memory cells for the GGNN enables better accuracy for the TCP use-case, while not providing better results in the UDP use-case.

Compared to the more simpler GNN-RNN architecture, we notice that we reach better accuracy using a GRU or a LSTM memory cell, even in case of memory stacks. This motivates our choice of using more recent graph-based neural network architectures from [4] compared to the earlier results from [5,6].

In order to better understand the difference between both GRU- and LSTM-based architectures, we investigate the average residual as illustrated in Fig. 8.

For both use-cases we notice similar values for normalized values below 0.5, where both architecture result in similar residual. In the TCP use-case, both architectures provide similar results across the range of measured values, with a tendency for the GRU-based architecture to underestimate bandwidths, while the GGNN-based one overestimates bandwidths. A similar behavior is exhibited for the UDP use-case.

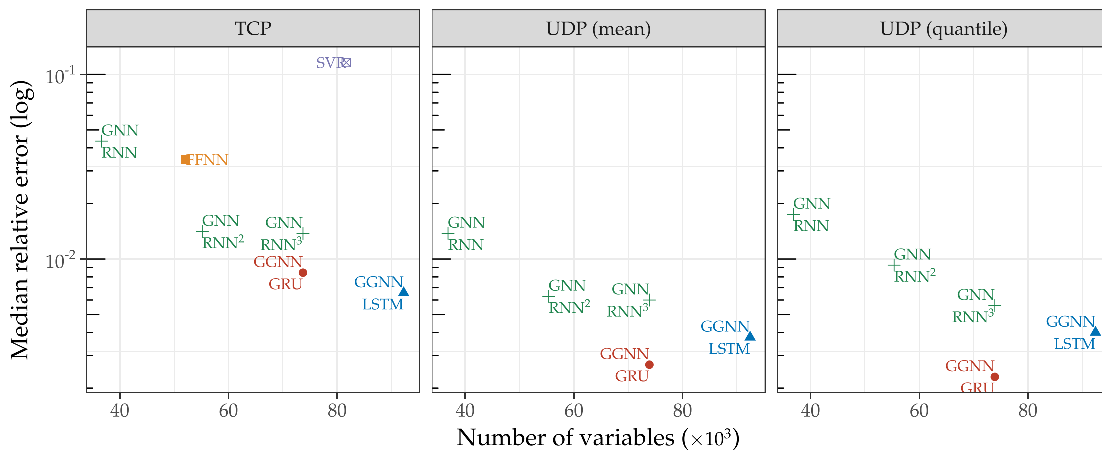


Fig. 9. Evaluation of the precision of the architectures against their number of variables.

5.5. *Influence of model size*

An important factor when comparing machine learning models and neural network architectures is their number of variables according to their precision, as illustrated in Fig. 9. For the SVR model, we quantify the number of variables as the size of the support vectors. For the neural network architectures, the number of variables presented in here are based on the same size for the hidden node representation (as highlighted in Table 1). Since model size directly correlates with training time and evaluation time, an ideal model would be in the bottom left part of Fig. 9, meaning it would provide the best accuracy with the lowest possible number of variables.

The difference between SVR and GGNN is also noticeable here, since both types of models use a similar number of variables, but with large differences in relative error. Fig. 9 illustrates also the motivation for using GGNNs with memory cells as the GGNN-GRU and GGNN-LSTM architectures outperform the other methods.

We also notice that although the GNN-RNN^3 and GGNN-GRU architectures have similar number of variables, the GGNN-GRU architecture outperforms the GNN-RNN^3 one, motiving our choice of more advanced graph-based neural network architecture.

5.6. *Influence of graph size*

We investigate in this section the relationship between size of the network topology and the precision of the model. Fig. 10 illustrates the influence of input graph size against the relative error. While some correlation between graph size and relative error are visible in Fig. 10, it is dependent on which dataset is investigated. We note that for the TCP use-case, increase in graph sizes result in an increase in the relative error. In the UDP use-case, the relationship between graph size and relative error is reversed.

5.7. *Influence of number of unrolled loops*

We noted in Section 3.2 that GGNNs are unrolled a fixed number of iterations T compared to GNNs. Fig. 11 illustrates the influence of T against the relative error, where the illustrated GGNNs were trained with different values for T. Since increasing T also has an impact on training and evaluation time, a trade-off between accuracy and speed has to be established when designing the graph neural network's architecture and choosing its hyper-parameters. Increasing T improves the prediction accuracy in both use-cases for the studied architectures, with smaller effect for T larger than 12, motivating our choice of $T = 12$ as presented in Table 1.

We notice that the UDP use-case highlights a larger difference between both architectures, where larger values of T for the GGNN-GRU provide only minimal impact on the prediction accuracy. This can be explained by the fact that end-to-end latencies depend more on local queuing effects than topology-wide ones. This is opposed to the performance of TCP flows, where the interaction between flows on a larger scale across the topology have larger impact than local effects.

5.8. *Execution time*

In order to understand the scalability of our model presented in Section 3, we evaluate in this section our implementation in terms of execution time. Since our datasets with TCP or UDP flows contain mainly small networks, we evaluated our

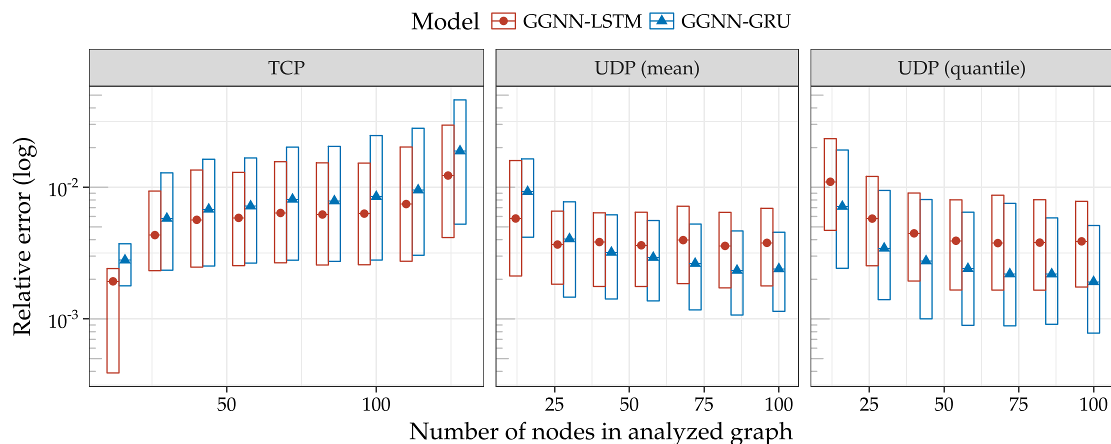


Fig. 10. Evaluation of architecture precision against the size of the studied graph for the GGNN-GRU and GGNN-LSTM models. Bars represent respectively the 25, 50 and 75 percentile values.

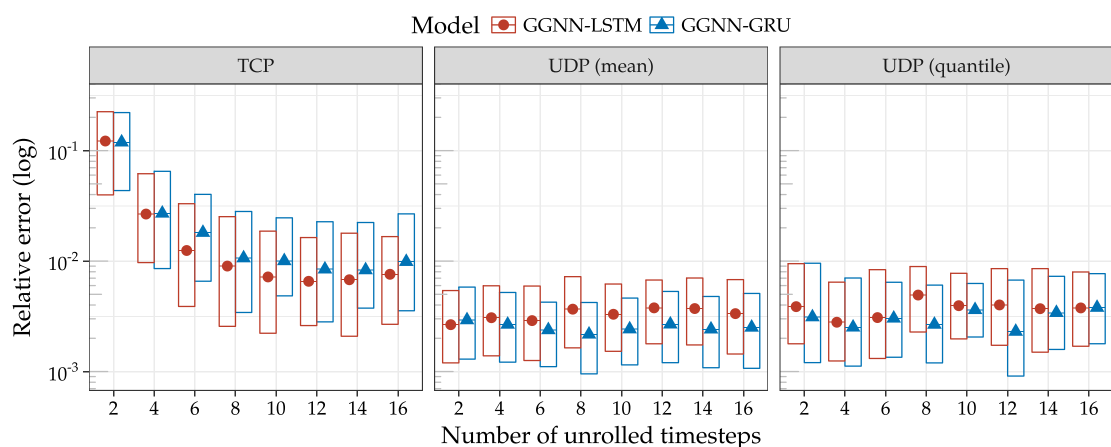


Fig. 11. Evaluation of architecture precision against the number of unrolled iterations of hidden representation propagation. Bars represent respectively the 25, 50 and 75 percentile values.

implementation against randomly generated graphs. We followed the Erdős–Rényi model [38] for generating random graph with a randomly selected number of nodes $|\mathcal{V}|$ following the uniform distribution $\mathcal{U}(2, 1000)$ and a random number of edges $|\mathcal{E}|$ following the uniform distribution $\mathcal{U}\,(N-1, D|\mathcal{V}|(|\mathcal{V}|-1)/2)$ with D the graph density parameter. The average graph density for both datasets used in Sections 5.2 and 5.3 was approximately 0.2.

Since the propagation of hidden node representations defined in Eqs. (2) and (6) may either be implemented in Tensorflow using dense matrix multiplication, or using sparse operations as noted by Allamanis et al. [20], we evaluated a dense and a sparse version of our model. According to the mathematical operations illustrated in Section 3 the runtime of one loop unrolling of the GNN correspond to the sum of the following terms:

- Per-node operations, namely Eqs. (7)–(10) for the GGNN-GRU, which scale linearly in execution time with the number of nodes, namely $\mathcal{O}\,(|\mathcal{V}|)$;
- The propagation of hidden node representations defined in Eqs. (2) and (6):
  - scaling linearly with the number of edges with sparse operations, namely $\mathcal{O}\,(|\mathcal{E}|)$, i.e. $\mathcal{O}\left(D|\mathcal{V}|^2\right)$ following that the $\max\,(|\mathcal{E}|) = D|\mathcal{V}|(|\mathcal{V}|-1)/2$,
  - scaling according to the multiplication of the hidden node representations with the graph adjacency matrix of size $|\mathcal{V}| \times |\mathcal{V}|$ with dense operations, namely $\mathcal{O}\left(\mathcal{H}|\mathcal{N}|^2\right)$ with $\mathcal{H}$ the size of hidden representation.

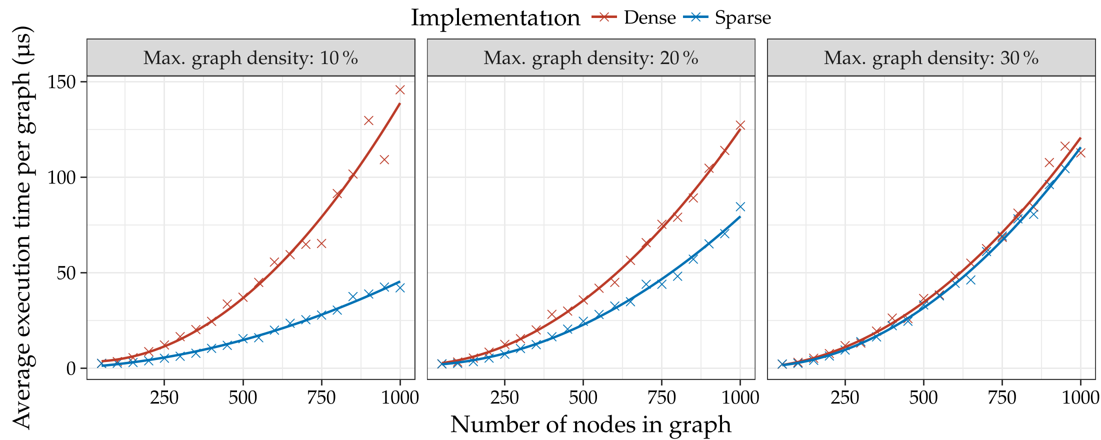


Fig. 12. Average execution time of prediction against number of nodes in graph. The fitted lines correspond to a polynomial model of order 2.

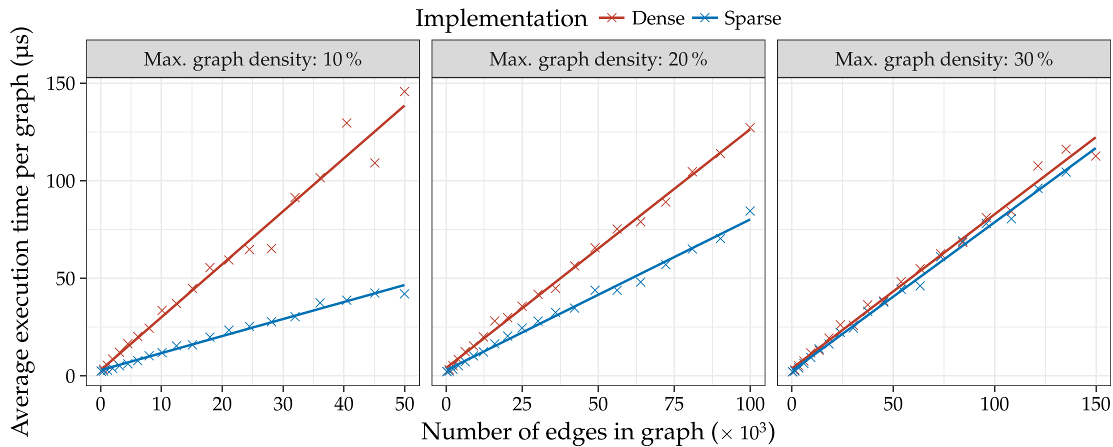


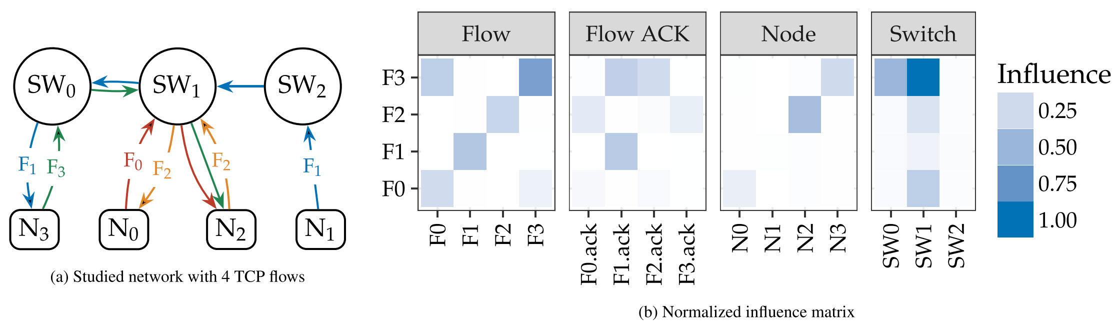
Fig. 13. Average execution time of prediction against number of edges in graph. The fitted lines correspond to a linear model.

Those operations are performed multiple times according to the number of loop unrolling, as discussed in Sections 3.2 and 5.7. In summary, our implementations should scale in execution time quadratically with the number of nodes in the graph, and linearly with the number of edges.

Numerical results are presented in Figs. 12 and 13 for our two different implementations and three values for the maximal graph density. The execution time measurements were measured on a server equipped with an Intel Xeon CPU E5-2620 v4 at 2.10 GHz and a Nvidia GeForce GTX 1080Ti. Batching of graphs was used to take advantage of parallelization. Those measurements validate the theoretical results presented earlier and illustrate the difference between the two implementations. As expected, the implementation using sparse operations performs faster for small graph densities while the implementation using dense operations performs the same regardless of graph density.

Those numerical results illustrate also applicability in practical use-cases since the execution time for prediction is below 200 us even for graphs with 1000 nodes and 150 000 edges. This small execution time enable fast operation in real networks, where the predictions may be used to dynamically configure computer networks.

Finally, while we illustrated in Figs. 12 and 13 the execution time for making a prediction on a given graph, our approach may still be limited by the size of the required dataset for larger networks in the training phase. Depending on the task which needs to be predicted, larger datasets may be required on larger graphs, since those larger cases may exhibit more complex behaviors. In the two use-case described in Sections 5.2 and 5.3, larger networks and larger number of flows will result in edge cases for the performance, namely low bandwidth for the TCP use-case and high latencies for the UDP one. Since those performance are inherently limited by link speed and maximum buffer sizes of the underlying topology in those edge cases, training and prediction on larger networks should still possible with moderate dataset sizes.



(a) Studied network with 4 TCP flows

(b) Normalized influence matrix

Fig. 14. Influence of nodes, other flows, and flow acknowledgments on the flows' performance.

6. Interpreting GNNs

An important subject when working with neural network is the interpretability of the learned model and its associated weights. A method often used for this purpose in computer vision is feature visualization using optimization [39], where inputs are optimized such that a specific output feature is maximized.

In our case, we are interested in visualizing the interaction between selected flows and the traversed queues. Once the parameters of the graph neural network learned, we replace the adjacency $\mathbf{A}$ with a modified matrix $\widetilde{\mathbf{A}}$ such that

$$\widetilde{\mathbf{A}} = \mathbf{A} \odot \sigma \left(\mathbf{B} + \mathbf{B}^T\right) \tag{28}$$

with $\mathbf{B} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ a learnable matrix. $\mathbf{B} + \mathbf{B}^T$ ensures that $\widetilde{\mathbf{A}}$ is a symmetrical matrix. For a given flow f, we then minimize the following loss function:

$$\left(\mathbf{y}_f - \mathbf{o}_f\right)^2 + \frac{1}{|\mathcal{V}|^2} \sum_{i,j} \widetilde{\mathbf{A}}_{\{i,j\}} \tag{29}$$

Fig. 14 illustrates the result of this optimization on a small topology with four TCP flows. We optimized $\mathbf{B}$ for each flow in the topology using gradient based optimization. The results are then aggregated according to the nodes in the network topology instead of the respective edges. The influence matrix corresponds to the aggregated value of $\mathbf{B}$.

As expected, bottlenecks and flows sharing the same bottleneck have an influence on each other, as shown for instance on the influence from SW_1 and F_0 on F_3. Nodes having no influence on the performance of flows can also be found, as illustrated here by SW_2 having no influence on F_1. Using this visualization method, root-cause analysis of poor performance and bottleneck identification may be performed.

7. Conclusion

We presented in this article *DeepComNet*, a novel approach for the performance evaluation of network topologies and flows based on graph-based deep learning. Our approach is based on the use of Gated Graph Neural Networks and a low-level graph-based representation of queues and flows in network topologies. Compared to other approaches using machine learning for performance evaluation of computer networks, the trained model is not specific to a given topology and high-level input features requiring more advanced knowledge on the studied protocol are not required.

We applied our approach to the performance evaluation of the bandwidth TCP flows and the performance evaluation of end-to-end latencies of UDP flows For both tasks, taking into account the topology in the performance evaluation is important: the throughput of TCP flows is dependent on the network architecture and network conditions (*i.e.* congestion and delays), and the end-to-end latencies depend on local queuing effects across the different queues from the topology. We showed via a numerical evaluation that our approach is able to reach good accuracies, with a median absolute relative error below 1%, even on large network topologies with multiple hops. We compared the chosen neural network architecture against two other approaches based on machine learning using high-level input features, and showed that our method outperforms those approaches by as much as one order of magnitude. Different types of GGNNs and configuration parameters were evaluated in order to understand which GGNN architecture produces the best results according to the studied network protocol. Finally we also gave some insights into the interpretation of the neural network in order to see which nodes in a given topology have an influence on protocol performances.

Since the network topology is directly taken as input of the neural network, applications such as network planning and architecture optimization may benefit from the method developed in this article. Future work may include further study of end-to-end protocols performance as well as optimization techniques based on this method.

Acknowledgments

We would like to thank Benedikt Jaeger for his feedback on an early draft of the paper as well as the anonymous reviewers for their valuable feedback. This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16KIS0538 (DecADe).

References

[1] G. Tian, Y. Liu, Towards Agile and smooth video adaptation in dynamic HTTP streaming, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, in: CoNEXT'12, ACM, ISBN: 978-1-4503-1775-7, 2012, pp. 109–120, http://dx.doi.org/10.1145/2413176.2413190.

[2] M. Mirza, J. Sommers, P. Barford, X. Zhu, A machine learning approach to TCP throughput prediction, IEEE/ACM Trans. Netw. (ISSN: 1063-6692) 18 (4) (2010) 1026–1039, http://dx.doi.org/10.1109/TNET.2009.2037812.

[3] M.B. Tariq, K. Bhandankar, V. Valancius, A. Zeitoun, N. Feamster, M. Ammar, Answering "What-If" deployment and configuration questions with WISE: techniques and deployment experience, IEEE/ACM Trans. Netw. (ISSN: 1558-2566) 21 (1) (2013) 1–13, http://dx.doi.org/10.1109/TNET.2012.2230448.

[4] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, in: Proceedings of the 4th International Conference on Learning Representations, in: ICLR'2016, 2016.

[5] M. Gori, G. Monfardini, F. Scarselli, A new model for learning in graph domains, in: Proceedings of the 2005 IEEE International Joint Conference on Neural Networks, in: IJCNN'05, vol. 2, IEEE, 2005, pp. 729–734, http://dx.doi.org/10.1109/IJCNN.2005.1555942.

[6] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2009) 61–80, http://dx.doi.org/10.1109/TNN.2008.2005605.

[7] Z.M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, K. Mizutani, State-of-the-art deep learning: evolving machine intelligence toward tomorrow's intelligent network traffic control systems, IEEE Commun. Surv. Tutor. (2017) http://dx.doi.org/10.1109/COMST.2017.2707140.

[8] H. Hours, E.W. Biersack, P. Loiseau, A causal approach to the study of TCP performance, ACM Trans. Intell. Syst. Technol. 7 (2) (2016) http://dx.doi.org/10.1145/2770878, 25:1–25:25.

[9] J. Padhye, V. Firoiu, D.F. Towsley, J.F. Kurose, Modeling TCP Reno performance: A simple model and its empirical validation, IEEE/ACM Trans. Netw. (ISSN: 1063-6692) 8 (2) (2000) 133–145, http://dx.doi.org/10.1109/90.842137.

[10] J. Bruna, W. Zaremba, A. Szlam, Y. LeCun, Spectral networks and locally connected networks on graphs, in: Proceedings of the 2nd International Conference on Learning Representations, in: ICLR'2014, 2014.

[11] M. Henaff, J. Bruna, Y. LeCun, Deep Convolutional Networks on Graph-Structured Data, arXiv:1506.05163.

[12] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, Computational capabilities of graph neural networks, IEEE Trans. Neural Netw. 20 (1) (2009) 81–102, http://dx.doi.org/10.1109/TNN.2008.2005141.

[13] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder–decoder for statistical machine translation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, 2014, pp. 1724–1734, http://dx.doi.org/10.3115/v1/D14-1179.

[14] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: Proceedings of the 5th International Conference on Learning Representations, in: ICLR 2017, 2017.

[15] M. Schlichtkrull, T.N. Kipf, P. Bloem, R.v.d. Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, in: Proceedings of the 15th European Semantic Web Conference, in: ESWC 2018, 2018.

[16] P.W. Battaglia, J.B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, R. Pascanu, Relational inductive biases, deep learning, and graph networks.

[17] A. Grover, J. Leskovec, node2vec: Scalable feature learning for networks, in: Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 855–864, http://dx.doi.org/10.1145/2939672.2939754.

[18] D. Marcheggiani, I. Titov, Encoding sentences with graph convolutional networks for semantic role labeling, in: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2017, pp. 1506–1515, http://dx.doi.org/10.18653/v1/D17-1159.

[19] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, G.E. Dahl, Neural message passing for quantum chemistry, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 1263–1272.

[20] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, in: Proceedings of the 6th International Conference on Learning Representations, in: ICLR 2018, 2018.

[21] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, D.L. Dill, Learning a SAT Solver from Single-Bit Supervision.

[22] M. Mathis, J. Semke, J. Mahdavi, T. Ott, The macroscopic behavior of the TCP congestion avoidance algorithm, ACM SIGCOMM Comput. Commun. Rev. 27 (3) (1997) 67–82, http://dx.doi.org/10.1145/263932.264023.

[23] N. Cardwell, S. Savage, T. Anderson, Modeling TCP latency, in: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, in: INFOCOM 2000, vol. 3, IEEE, 2000, pp. 1742–1751, http://dx.doi.org/10.1109/INFCOM.2000.832574.

[24] V. Firoiu, I. Yeom, X. Zhang, A framework for practical performance evaluation and traffic engineering in IP networks, in: Proceedings of the IEEE International Conference on Telecommunications, 2001.

[25] P. Velho, L.M. Schnorr, H. Casanova, A. Legrand, Flow-level network models: have we reached the limits?, Tech. Rep. 7821, INRIA, 2011.

[26] F. Geyer, S. Schneele, G. Carle, Practical performance evaluation of ethernet networks with flow-level network modeling, in: Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools, in: VALUETOOLS 2013, 2013, pp. 253–262, http://dx.doi.org/10.4108/icst.valuetools.2013.254367.

[27] F. Geyer, Performance evaluation of network topologies using graph-based deep learning, in: Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools, in: VALUETOOLS 2017, 2017, pp. 20–27, http://dx.doi.org/10.1145/3150928.3150941.

[28] L.B. Almeida, in: J. Diederich (Ed.), Artificial neural networks, IEEE Press, ISBN: 0-8186-2015-3, 1990, pp. 102–111.

[29] F.J. Pineda, Generalization of back-propagation to recurrent neural networks, Phys. Rev. Lett. 59 (1987) 2229–2232, http://dx.doi.org/10.1103/PhysRevLett.59.2229.

[30] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Netw. Mach. Learn. 4 (2) (2012) 26–31.

[31] S. Hochreiter, J. rgen Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780, http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[32] R. Pascanu, C. Gulcehre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks, in: Proceedings of the 2nd International Conference on Learning Representations, in: ICLR 2014, 2014.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L.u. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017, pp. 6000–6010.
[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, URL http://tensorflow.org/, Software available from tensorflow.org, 2015.
[35] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.
[36] S. Semeniuta, A. Severyn, E. Barth, Recurrent Dropout without Memory Loss, arXiv:1603.05118.
[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[38] P. Erdős, A. Rényi, On random graphs. I, Publ. Math. 6 (1959) 290–297.
[39] M.D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, in: D. Fleet, T. Pajdla, B. Schiele, T. Tuytelaars (Eds.), Computer Vision – ECCV 2014, Springer International Publishing, ISBN: 978-3-319-10590-1, 2014, pp. 818–833, http://dx.doi.org/10.1007/978-3-319-10590-1_53.

Fabien Geyer is currently with Airbus Central Research & Technologies working on methods for network analytics, network performances and architectures. He received the master of engineering in telecommunications from Telecom Bretagne, Brest, France in 2011 and the Ph.D. degree in computer science from Technische Universität München (TUM), Munich, Germany in 2015. His research interests include formal methods for the performance evaluation and modeling of network architectures and protocols.

A.2.3 DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning

This work was published in *IFIP Networking 2019*, 2019 [70].

DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning

Fabien Geyer
Technical University of Munich & Airbus CRT, Germany

Stefan Schmid
Faculty of Computer Science, University of Vienna, Austria

***Abstract*—With the increasing complexity of communication networks and the resulting threat of disruptions of mission critical services due to manual misconfiguration, automated verification is becoming a key element in today's network operation. In particular, it has recently been shown that a polynomial-time, automated verification of the policy-compliance of network configurations is possible for the important class of MPLS networks, even under failures. However, this approach, while providing polynomial runtimes, is still fairly slow in practice and only allows to *detect* but not *fix* configurations.**

This paper proposes a novel approach to speed up the analysis of network properties as well as to suggest configuration changes in case a network property is not satisfied. More specifically, our solution, *DeepMPLS*, allows to predict if a network property is satisfiable, and if not, aims to present a counter example. We also show that *DeepMPLS* may be used to propose new prefix-rewriting rules in the MPLS configuration in order to make it satisfiable. *DeepMPLS* can hence be used for fast predictions, before more rigorous analyses are performed.

***DeepMPLS* is based on a new extension of graph-based neural networks. Our prototype implementation, using Tensorflow, achieves low execution times and high accuracies in real-world network topologies.**

I. Introduction

As communication networks are increasingly used for critical services such as health monitoring, power grid management, or disaster response [1], their uninterrupted availability is more important than ever before. However, the increasing dependability requirements stand in stark contrast to today's manual approach to operate networks with often very complex configurations.

Automated approaches can greatly improve the trustworthiness of networks and hence reliability, by allowing to test a large number of network configuration for their policy compliance. Yet, many network verification tools still require a super-polynomial runtime to test basic connectivity properties [2, 3, 4]. Testing whether network configurations are policy compliant even under failures, introduces another combinatorial complexity.

It was recently shown that for the widely deployed MPLS networks, a polynomial-time "what-if analysis" is possible [5]: an automata-theoretic approach, leveraging a connection to prefix rewriting systems, can be used to test important properties such as connectivity (can two endpoints reach each other?), loop-freedom (may packets be forwarded in circles?) or waypoint enforcement (is traffic always going through the firewall?), even under failures. While this is promising, the runtime in practice is still relatively high (in the order of an hour even for relatively small yet complex networks): essentially, the approach in [5] requires the construction of a large pushdown automaton (PDA), based on the network configuration, the routing tables, as well as the query; the PDA is then solved using reachability analysis. While PDA is still polynomial in size, it can quickly grow to millions of nodes and transitions in realistic networks, for which reachability has to be solved *for each query*. Furthermore, the approach can only be used to *verify* properties, but not to *repair* configurations, e.g., to re-establish invariants.

This paper is motivated by the question whether it is possible to build upon these recent results while exploiting opportunities for speeding up verification as well as to support an automated fixing of configurations. This is challenging also because unlike other networks, MPLS supports arbitrary (and in principle unbounded) header sizes: additional labels are for example pushed to route around railed links. Our work is also motivated by a novel approach which seems to fit the specific problem particularly well: Graph Neural Networks [6, 7] have already been applied successfully in many contexts, including molecule analysis [8, 9] or jet physics [10], but despite being a natural choice for our problem, its potential is largely unexplored.

A. Our Contributions

This paper presents a novel approach to speed up verification and synthesis of the policy-compliance of network configurations. At the heart of our our tool, *DeepMPLS*, lies a new extension of graph-based neural networks: leveraging deep learning, *DeepMPLS* allows to predict counter examples (i.e., "proofs" or witness traces) to specific network properties (or queries), which can be verified fast. In fact, in this paper we show that *DeepMPLS*'s probabilistic approach may even be used for *synthesis*: it has the potential to predict which MPLS rules should be added, in order to *re-establish* certain properties. The tool may hence overcome the need to perform more rigorous and time-consuming analyses in many scenarios.

Our experiments, using our TensorFlow prototype implementation, show promising results: on real network topologies, *DeepMPLS* can achieve a high degree of accuracy with fast execution time.

ISBN 978-3-903176-08-9 ©2019 IFIP

As a contribution to the research community and in order to ease future research, our dataset and experimental data used in this paper is also available online.

B. Scope and State-of-the-Art

We are concerned with the correct, i.e., policy-compliant configuration of widely deployed MPLS networks. In particular, we are interested in predicting and fixing *properties* of MPLS networks which are described as a regular query language, as it is also used in the state-of-the-art *P-Rex* tool [4] motivating our paper. To this end, a formal model for MPLS networks is required. Before we sketch the model (see [4] for more details), we briefly review some basic concepts of MPLS networks.

In a nutshell, MPLS networks operate between Layer 2 and Layer 3 and rely on tunnels across a transport medium. Forwarding decisions are based on the *top-of-stack label*: an MPLS node (i.e., a.k.a. label switch router a.k.a. transit router) uses the top label of the label stack included in the packet header, to determine the next hop on a Label Switched Path (LSP). On this occasion, the old label can be replaced with a new label before the packet is routed forward. An MPLS node serving as *label edge router* acts as the entry and exit point for the network: a label edge router pushes an MPLS label onto an incoming packet resp. pops it from an outgoing packet.

More specifically, upon receiving a packet and depending on the content of the top of the stack label, an MPLS node performs a *swap*, *push* or *pop* operation on the packet label stack: In a *swap* operation the label is swapped with a new label, and the packet is forwarded along the path associated with the new label; in a *push* operation a new label is pushed on top of the existing label, encapsulating the packet in another layer of MPLS and introducing hierarchical routing; and in a *pop* operation the label is removed from the packet. If the popped label was the last on the stack, the packet leaves the MPLS tunnel.

In order to deal with failures, MPLS includes a local protection mechanism allowing to protect a label switched path by a backup path. This mechanism is based on the recursive pushing of labels, i.e., tunnels, around a failed link.

Our work is motivated by the goal to predict and fix properties according to a natural regular query language [4]. A (reachability) query is of the form $<a> \;\; b \;\; <c> \;\; k$ where the regular expression a describes the (potentially infinite) set of allowed initial label-stack headers, the regular expression b describes the set of allowed routing traces through the network, and the regular expression c describes the set of label-stack headers at the end of the trace. Finally, k is a number specifying the maximum allowed number of failed links.

This query needs to be answered for a given MPLS network whose configuration can be described as a tuple $N = (V, I, L, E, \tau)$: V is the set of routers, I the set of all interfaces in the network connected by links E, L the set of label stack symbols and τ the routing table (including conditional failover rules).

P-Rex allows to answer the following question in polynomial time: is there a set of failed links F with $|F| \leq k$ for a given network configuration $N = (V, I, L, E, \tau)$ such that there results a route (i.e., *trace*) satisfying the regular expression?

Towards this goal, P-Rex automatically collects the current routing tables, and given them as well as the network and the query, constructs a pushdown automaton (PDA) on which reachability analysis is performed using the Moped tool. More specifically, the initial header and final header regular expressions of the query are first converted to a Nondeterministic Finite Automaton (NFA) and then to a Pushdown Automaton (PDA). The path query is converted to an NFA, which is used to augment the PDA constructed based on the network model. The three PDAs are combined into a single PDA which simulates the automata running in lockstep and can then be queried.

P-Rex then not only provides a yes or no answer, but also a witness, if it exists. Furthermore, P-Rex additional optimizations such as "top of stack reduction", reducing the number of transitions in the PDA.

Thus, the challenge considered in this paper is to not only reduce the runtime further (which keeping the support for arbitrary header sizes and multiple link failures), using a novel methodology, but also to again "*synthesize*" a witness for each query. In particular, we would like to improve the runtime for "hard queries": to this end, we propose a methodology which considers the size of the PDA as a measure of complexity

C. Organization

The remainder of this paper is organized as follows. In Section II, we present our approach and solution in detail. Section III describes the dataset used for training and evaluating our approach, and Section IV reports on our experimental results. After reviewing related work in Section V, we conclude our work in Section VI and discuss future work.

II. DeepMPLS based on Graph Neural Network

This section presents our approach, *DeepMPLS*, supporting the fast testing and synthesis of MPLS network configurations. The main idea behind *DeepMPLS* is to map network topologies to graphs, which can then be processed using neural networks. Our graph representation has nodes representing routers, physical interfaces of routers and additional nodes used for describing MPLS configurations and queries. Edges between the nodes represent physical links in the topology, as well as the interactions between a MPLS configuration and elements of the network topology. Those graphs are then used as input for a neural network architecture able to process general graphs.

Compared to other representations used in machine learning which require to summarize properties of a topology in a vector of fixed size, our approach is not limited by the size of the topology or its configuration. This means that an accurate description of the complete topology and its configuration can be passed to the neural network.

A. Leveraging Graph Neural Networks

The neural network architecture used by *DeepMPLS* is based on an extension of Graph Neural Networks [6, 7]. In the following, let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and target values of node v.

Graph Neural Networks rely on a *message passing* concept:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}_u^{(t-1)} \,\middle|\, u \in \textsc{Nbr}(v)\right\}\right) \quad (1)$$

$$\mathbf{o}_v = g\left(\mathbf{h}_v^{(t\to\infty)}\right) \quad (2)$$

$$\mathbf{h}_v^{(t=0)} = init\left(\mathbf{i}_v\right) \quad (3)$$

with $\mathbf{h}_v^{(t)}$ corresponding to the hidden representation of node v at time t, $f(\cdot)$ a function which aggregates the hidden representations, $\textsc{Nbr}(v)$ the set of neighboring nodes of v, $g(\cdot)$ a function transforming the final hidden representation to the target values, and $init(\cdot)$ an initialization function for the hidden representations.

The concrete formulations of the *aggr* and *out* functions are feed-forward neural networks (FFNN), with the addition that *aggr* is the sum of per-edge terms [7], such that:

$$\mathbf{h}_v^{(t)} = aggr\left(\left\{\mathbf{h}_{\textsc{Nbr}(v)}^{(t-1)}\right\}\right) = \sum_{u\in\textsc{Nbr}(v)} f\left(\mathbf{h}_u^{(t-1)}\right) \quad (4)$$

with f a FFNN. *init* is modeled as a one-layer FFNN which produces a vector respecting the dimensions of the hidden representations.

Gated Graph Neural Networks (GGNN) [11] were recently proposed as an extension of GNNs to improve their training. This extension implements f using a memory unit called Gated Recurrent Unit (GRU) [12] and unrolls Equation (1) for a fixed number of iterations. This simple transformation allows for commonly found architectures and training algorithms for standard FFNNs as applied in computer vision or natural language processing.

Formally, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \quad (5)$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z\mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \quad (6)$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r\mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \quad (7)$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W}x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \quad (8)$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \quad (9)$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}$ and $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}$ are trainable weight matrices, and $\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining the edges in the graph $\mathcal{G}$.

Equation (5) corresponds to one time-step of the propagation of the hidden representation of neighboring nodes to node v, as formulated previously for GNNs in Equations (1) and (4). Equations (6) to (9) correspond to the mathematical formulation of a GRU cell [12], with Equation (6) representing the GRU reset gate vector, Equation (7) the GRU update gate vector, and Equation (9) the GRU output.

B. Application to MPLS Network Analysis

In order to tailor the above concepts to MPLS network verification and synthesis, we need a transformation of network topology and MPLS configuration to a graph. The transformation process we propose in this paper is illustrated in Figures 1 to 4, where the MPLS network depicted in Figure 1 is transformed into a graph.

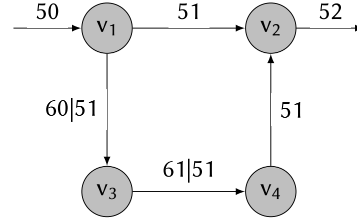


Figure 1: Example MPLS network. In case the link between v_1 and v_2 fails, a backup tunnel (v_1, v_3, v_4) has is used around the failed link.

Each router $v \in V$ in the network is represented as a node in the graph $\mathcal{G}$. Each network interface $i \in I_v^{in} \cup I_v^{out}$ is also represented as a node, connected via an edge to its router. Links in the topology E are represented as edges connecting the two corresponding network interfaces.

As presented in Figures 2 and 3, the MPLS configuration of each router is also encoded as nodes and edges in the graph. Each MPLS label $l \in L$ is represented as a node. The routing table of each router $\tau_v : I_v \times L \to (2^{I_v \times Op^*})^*$ is represented as a set of rules. Each rule is represented as a node in the graph, connected the nodes representing its input interface $i \in I$ as well as its input label $l \in L$. The actions $o \in Op$ associated to a rule are also represented as nodes, connected via edges in case multiple actions are to be performed for a given rule as illustrated in Figure 3. MPLS actions with label parameters such as *swap* or *push* are connected to their respective label node. The last action associated to a rule is connected to its output interface.

Queries are also encoded as nodes in the graph as illustrated in Figure 4. In this paper, we will assume the same notation as [4] for specifying queries, namely the regular expressions which are defined over an alphabet Σ and use the abstract syntax

$$a ::= s \mid . \mid [\hat{}\, s_1, \ldots, s_n] \mid a_1 + a_2 \mid a_1 a_2 \mid a^*$$

where

s	is a symbol from Σ,
$.$	is a wildcard for any symbol from Σ,
$[\hat{}\, s_1, ..., s_n]$	stands for any symbol $s \in \Sigma \setminus \{s_1, \ldots, s_n\}$,
$a_1 + a_2$	is the choice between a_1 and a_2,
$a_1 a_2$	is the concatenation of a_1 and a_2, and

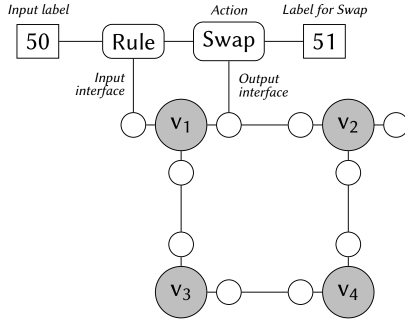


Figure 2: Encoding of network topology and a MPLS rule for v_1 as graph.

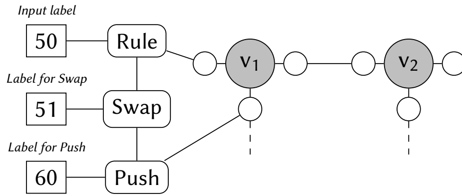


Figure 3: Encoding of network topology and a second MPLS rule for v_1 as graph.

a^* is the concatenation of 0 or more occurrences of a.

The set of all regular expressions over Σ is denoted by $Reg(\Sigma)$ and we assume a standard definition of the language $Lang(a) \subseteq \Sigma^*$ that is described by a regular expression a.

We follow an approach inspired by the McNaughton-Yamada-Thompson algorithm [13] which transforms a regular expression into an equivalent nondeterministic finite automaton. The different symbols of the regular expression of a query are represented as nodes, with edges representing their relationships. In case a symbol corresponds to a MPLS label or a router in the network, we reuse the node which was already defined in the graph. Wildcard symbols are represented as special nodes in the graph as illustrated in Figure 4.

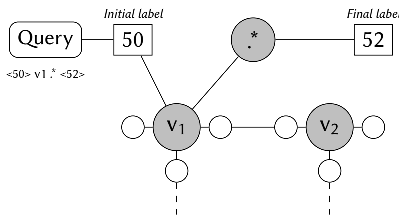


Figure 4: Encoding of network and query as graph.

Relationship between symbols such as combinations ($a_1 + a_2$) are represented using edges in the query representation, as illustrated in Figure 5.

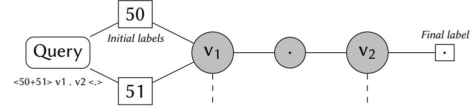


Figure 5: Encoding of more complex query as graph.

Each node in the graph may have input features describing characteristics of the node. In our case, nodes are mainly represented by their type, encoded as categorical value. We define the 12 following types for the nodes:

- Elements of the network topology: *Router*, *Interface*;
- Elements of the MPLS configuration: *Rule*, *Label*, *Push Action*, *Swap Action*, *Pop Action*;
- Elements of the query and the regular expression: *Query*, *Label Wildcard*, *Label Dot*, *Router Dot*, *Router Wildcard*.

Additionally to the type, the node representing a query has an additional input feature corresponding to the k parameter. Edges in the graph have no input features and represent only the relationship between nodes.

For training of the graph neural network, we use different output features depending on the prediction which is required. We define the three following prediction tasks:

Satisfiability Heuristic for verifying if a query is satisfiable.

Routing trace Heuristic for generating a trace of routers which match a satisfiable query.

Partial synthesis Synthesis of an MPLS configuration in order to satisfy a query.

Example queries for those three tasks are detailed later in Section III.

For the *Satisfiability* task, a classification task is defined where the query node is classified in two categories (*true* or *false*) in case the query is satisfiable or not. The training is done using a softmax cross entropy loss on the query node.

Similarly, for the *Routing trace* task, router nodes are classified in two classes, namely if the router is part of the trace or not. Since we are interested in the per-topology prediction, namely correctly classifying the router nodes in the graph, the training is done using a graph-wide sigmoid cross entropy loss.

Finally for the *Partial synthesis* task, missing rule and action nodes are added in the graph with the goal of connecting them via edges to the appropriate router, label and interfaces in order to satisfy a query. This case is an edge prediction task. Predicted edges in the graph are then used for reconstructing the MPLS prefix-rewriting rules. A similar loss function than for the *Routing trace* task is used here.

C. Complexity Analysis

In order to understand the scalability of the model presented in Section II-A, we evaluate the complexity of the algorithm. According to the mathematical operations illustrated earlier, the runtime of one loop unrolling of the GNN corresponds to the sum of the following terms:

- Per-node operations, namely Equations (6) to (9), which scales linearly in execution time with the number of nodes, namely $\mathcal{O}(|\mathcal{V}|)$;

- The propagation of hidden node representations defined in Equations (1) and (5), which scales linearly with the number of edges, namely $\mathcal{O}(|\mathcal{E}|)$ if sparse operations are used (i.e. using the graph's adjacency list), or quadratically to the number of nodes if the graph's adjacency matrix is used instead $\mathcal{O}(|\mathcal{V}|^2)$.

In order to achieve good accuracy, the recursion from Equation (1) is unrolled for a fixed number of iterations according to $\mathcal{D}$, the diameter of the analyzed graph. In total, the runtime complexity is summarized as: $\mathcal{O}\left(\mathcal{D}(|\mathcal{E}| + |\mathcal{V}|)\right)$.

III. Methodology and Dataset Generation

In order to train our neural network architecture, we used the Topology Zoo [14] – a collection of over 250 networks used in real-file – as a basis for generating network topologies. Each network topology was either taken as is or randomly modified, either by removing a router, or by adding a router and connecting it to a set of randomly chosen routers.

Based on those network topologies, MPLS configurations were generated for each network topology. With a given probability, MPLS tunnels were constructed between randomly selected pairs of routers in the network using dedicated MPLS labels. Additionally to the tunnel corresponding to the shortest path in the topology, a random number of additional backup tunnels were also generated following different paths in the network when possible. MPLS label stacking was used in for those backup tunnels, following common practice for automatic fail-over.

For generating queries, we randomly generate regular expressions as follows, with l_i representing the input label, l_o to output label, r_i the input router, r_o the output router, and k the maximum allowed number of failed links:

- $< l_i > r_i < l_o > k$ is satisfied if a packet with label l_i, crosses router r_i, and exists with label l_o;
- $< l_i > r_i \ .^* \ r_o < l_o > k$ is satisfied if a packet with label l_i entering at router r_i, traverses an unknown number of other routers, and exists from router r_o with label l_o;
- $< l_i > . \ .^* \ r_o < l_o > k$ is satisfied if a packet with label l_i enters the network, traverses at least one router, and exists from router r_o with label l_o;
- $< .^* > r_i \ .^* \ r_o < l_o > k$ is satisfied if there is a path from router r_i to r_o where the output label is l_o;
- $< l_i > r_i \ .^* \ r_o < .^* > k$ is satisfied if a packet with label l_i entering at router r_i, traverses the network, reaches router r_o with label l_o.

Those queries, inspired by the ones presented in [4], mainly focusing on reachability.

Routers and labels are either select randomly in the set of available routers V and labels L – resulting in most cases in non-satisfiable queries – or they are selected such that the query is satisfiable. In order to generate those satisfiable queries, we construct a so-called *MPLS traversal graph* for each network topology. In such a directed tree, each node is a tuple of the form $(input\ label, router, output\ label(s))$ corresponding to the result of the MPLS routing table of each router in the topology. Based on those MPLS traversal graphs, satisfiable queries can be generated by randomly selecting a node in the graph and traversing it randomly. Finally the k parameter of the query is generated randomly following a discrete uniform distribution.

The satisfiability of each generated query is tested using P-Rex [4], which relies on the construction of a push-down automaton based on the query as well as the network. Concretely, P-Rex generates one big pushdown automaton (PDA) based on the regular expressions of the initial header and final header defined by the query (which are first converted to nondeterministic finite automata) as well as the nondeterministic finite automata describing the path query.

Typically, the larger the PDA, the higher is the runtime of reachability analysis, and accordingly, in our methodology in the following, we will interpret the size of the PDA as a measure of complexity. Accordingly, for each evaluation of P-Rex, the size of the generated push-down automaton is recorded and will serve as a numerical measure of the complexity of the query in Section IV.

P-Rex can directly be used for defining the required outputs of the *Satisfiability* and *Trace* tasks. For the *Partial synthesis* task, we first randomly generate a satisfiable query and randomly remove a maximum of two MPLS prefix-rewriting rules such that the query is not satisfiable anymore. The rules which trigger the loss of satisfiability are the rules that *DeepMPLS* has to predict.

In total, more than 90.000 topologies and queries were generated. Table I summarizes different statistics about the generated dataset. The dataset is available online[1] to reproduce the results.

Parameter	Min	Max	Mean	Median
# of routers	3	30	10.6	10
# MPLS labels	8	689	225.3	174
# MPLS rules	8	795	319.5	248
Size of push-down automaton	17	37006	5441.2	2692
# of nodes in analyzed graph	36	2333	914.4	713
# of edges in analyzed graph	48	4000	1615.4	1261

Table I: Statistics about the generated dataset.

IV. Evaluation

We evaluate in this section *DeepMPLS* on the three prediction tasks described in Section II-B against our dataset of topologies and queries. Following current best practices for machine learning, the dataset was randomly split in two parts: training (80 % of the topologies) and test (20 % of the topologies). The neural network was trained on the training dataset, while the evaluation and metric figures reported later in this section were computed using the test dataset. Due to the lack of availability of other topologies and their MPLS configuration, no validation dataset was used.

Via a numerical evaluation, we illustrate the accuracy and execution time of *DeepMPLS* and highlight its usability

[1] https://github.com/fabgeyer/dataset-networking2019

for practical use-cases. The performances are also compared against a simpler heuristic based on a random walk in the MPLS network, and random prediction of MPLS rules to add to the configuration.

A. Technical Implementation

We implemented *DeepMPLS* using TensorFlow. For the purpose of computational efficiency, sparse operations are used for passing of hidden representation between neighboring nodes. Table II illustrates the size of the different layers of the neural network used for the numerical evaluation.

The recursion from Equation (1) was unrolled for a fixed number of iteration according to the diameter of the analyzed graphs. A detailed evaluation of the importance of this number of iterations will be performed in Section IV-F.

Layer	**NN Type**	**Size**
init	FFNN	$(14, 80)_w + (80)_b$
Memory unit	GRU cell	$(160, 160)_w + (160, 80)_w + (240)_b$
Edge attention	FFNN	$(160, 1)_w + (1)_b$
out hidden layers	FFNN	$2 \times \{(80, 80)_w + (80)_b\}$
out final	FFNN	$(80, 2)_w + (2)_b$
	Total:	53 124 parameters

Table II: Size of the different layers used in the GGNN. Indexes represent respectively the weights (w) and biases (b) matrices.

B. Random Walk Baseline

In order to have a baseline for evaluating the accuracy of the predictions of the neural network for the *Satisfiability* and *Trace* tasks, we introduce here a simple heuristic based on random walks in the MPLS network.

This heuristic selects a starting router and label in the topology according to the first elements of the query, and traverses the network according to the MPLS prefix-rewriting rules until the destination specified by the query is reached. In case multiple rules apply to a given input label, a random rule is selected and its prefix-rewriting actions are applied. If the random walk was not successful, another random walk is performed until a maximum number of walks of 10 is reached.

Since our dataset also contains queries which do not explicitly specify the starting label or starting node, as explained in Section III, a random starting point in the network is selected which matches the explicit parts of the query in those cases.

On the generated dataset, this heuristic was able to predict the satisfiability of a query with an accuracy of 79.2 %. As illustrated later in Figure 8, this accuracy decreases with the complexity of the query.

C. Neural Network Training

We first evaluate the training of *DeepMPLS* for the *Satisfiability* task, namely prediction of the satisfiability of a query given an topology and MPLS configuration. Figure 6 illustrates the accuracy of *DeepMPLS* during training according to the number of training iterations, on both the training and the test dataset. Each training iteration corresponds to 16 analyzed topologies and queries from the training dataset, i.e. their representations as graphs. After 2500 training iterations, *DeepMPLS* reaches the accuracy of the baseline on the test dataset, before converging after around 25 000 training iterations.

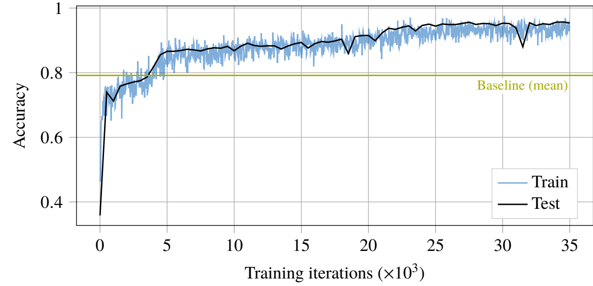


Figure 6: Training of *DeepMPLS* for prediction of query satisfiability and comparison against baseline.

Based on this first training, we retrain the same neural network and same set of weights for the *Routing trace* task, namely prediction of the routing trace of a query in case a query is satisfiable. This technique, also known as *knowledge transfer*, is often used in deep learning in order to accelerate training.

Figure 7 illustrates the accuracy of *DeepMPLS* on this second task according to the number of training iterations. Since the neural network was already pre-trained on the first task, this second training requires fewer iterations in order to reach good prediction accuracy. Less than 1000 iterations are required before convergence of the accuracy.

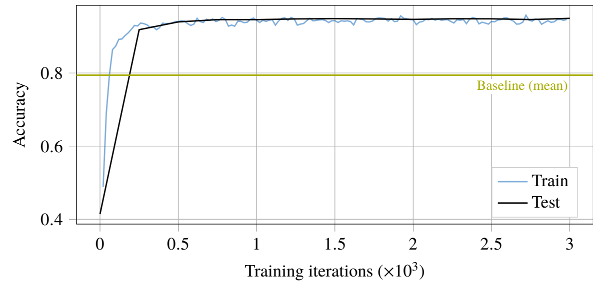


Figure 7: Training of *DeepMPLS* for trace generation using pre-trained weights.

The same approach of knowledge transfer was used for training *DeepMPLS* against the *Partial Synthesis* task, resulting in faster training convergence, with a training curve similar to the one illustrated in Figure 7.

D. Model Performance

We next assess the performance of *DeepMPLS* in the three different tasks presented in Section II-B.

1) Satisfiability task: We evaluate here the performance of *DeepMPLS* at predicting if a query is satisfiable or not. We use the prediction accuracy, precision and recall as metrics for evaluating the model.

Figures 8 and 9 illustrate the accuracies, precision and recall of *DeepMPLS* and the baseline. In average, *DeepMPLS* is able to predict the satisfiability of a query with an accuracy of 95.4 %, a precision of 97.9 %, and a recall of 89.2 %. While the performance of the baseline drops with the size of the push-down automaton, *DeepMPLS* still performs well on those more complex cases.

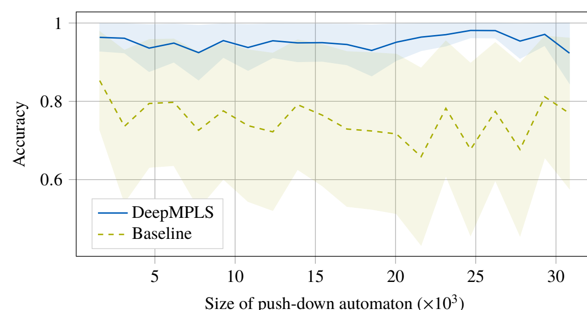


Figure 8: Accuracy of *DeepMPLS* and baseline against size of push-down automaton. Bands indicate the variance of the accuracy according to the push-down automaton sizes.

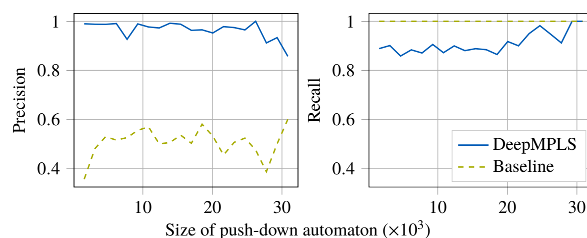


Figure 9: Precision and recall of *DeepMPLS* and baseline against size of push-down automaton.

2) Routing trace task: We evaluate here the performance of *DeepMPLS* at predicting the routing trace if a query is satisfiable. *DeepMPLS* was able to classify the routers with an overall accuracy of 92.8 % and a precision of 91.5 %. If we redefine the accuracy as the correct prediction of all routers in a given topology, this per-topology accuracy of *DeepMPLS* is of 68.2 % in average.

Figure 10 illustrates this per-topology accuracy, with the detail of true positives and true negative. *DeepMPLS* has good performance for the true negatives with an average of 99.4 %, while it reaches only a average of 85.6 % for the true positives. This means that routers in the true routing trace are sometimes missing in the prediction, but routers absent from the true routing trace are rarely selected (i.e. low false negative rate).

3) Partial synthesis task: Finally, we evaluate here the accuracy of for the *Partial synthesis* task, namely predicting

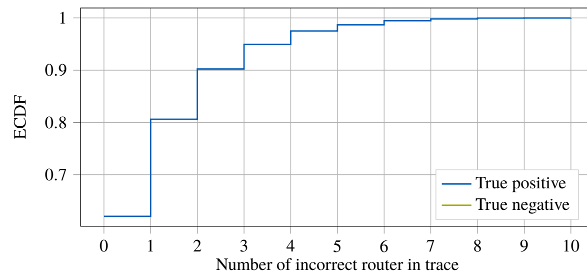


Figure 10: Per-topology accuracy of *DeepMPLS* with details on true and false positives.

which additional rules needs to be installed in a network in order to satisfy a query. Since the baseline described in Section IV-B cannot be applied here, we define a new one for this task. This new baseline randomly selects the requested number of edges in the *DeepMPLS* graph model following a simple random sampling without replacement.

Figure 11 illustrates the detailed per-topology accuracy of *DeepMPLS* and the baseline. In average, *DeepMPLS* is able to predict the correct edges with an accuracy of 45.9 %, while the random baseline predicts them with an average accuracy of only 0.1 %.

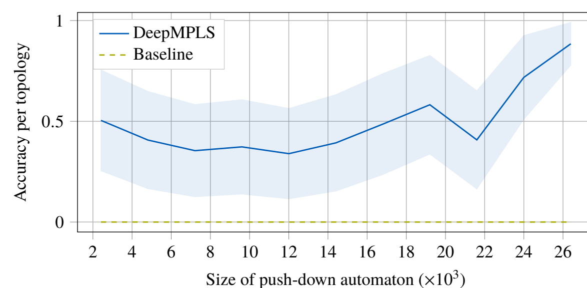


Figure 11: Per-topology accuracy of *DeepMPLS* and baseline.

E. Execution Time

In order to understand the practical applicability of *DeepMPLS*, we evaluate in this section its execution time in different settings. This part is a complement to the complexity analysis presented in Section II-C. We define and measure the execution time per network as the total time taken to process the network and a satisfiability query, without including the startup time or the time taken for initializing the network data structures.

Since the neural network can be evaluated on either CPU or GPU, we evaluated *DeepMPLS* on both platforms. A Nvidia GTX 1080 Ti was used for the measurements on GPU, and an Intel Xeon Gold 6130 CPU was used for the ones on CPU. The same CPU was used for the execution time measurement of P-Rex.

Figure 12 illustrates the different execution times and compares *DeepMPLS* against P-Rex. For the three different

evaluations, we note a linear relationship between size of the push-down automaton – and hence size of the analyzed graph – and the execution time. *DeepMPLS* is one order of magnitude faster than P-Rex when running on CPU, and two order of magnitudes faster on GPU, mainly due to the better ability of GPUs of parallelizing the numerical operations used in neural networks. Those figures illustrate that *DeepMPLS* show promising applicability due to fast computation times.

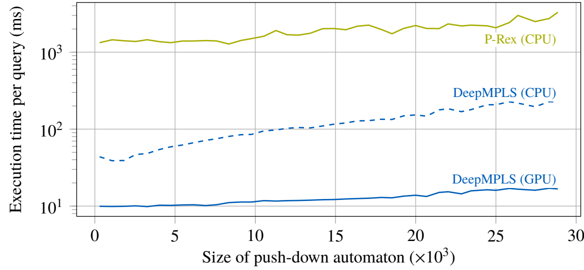


Figure 12: Execution time of *DeepMPLS* on CPU and GPU compared against P-Rex for the *Satisfiability* task.

F. Impact of Number of Iterations

We described in Section II-A that the GNN requires multiple evaluations of the recurrence defined in Equation (5) in order to propagate the hidden representations across multiple hops. We evaluate in this section the relationship between the number of iterations and the prediction accuracy of *DeepMPLS*.

Numerical results are illustrated in Figure 13 for the *Satisfiability* task. As the number of iterations increases, the prediction accuracy also increases. Convergence is reached after approximately 16 iterations. Since this parameter directly influences the execution time of *DeepMPLS*, a proper value has to be chosen in order to have a good trade-off between accuracy and computational complexity.

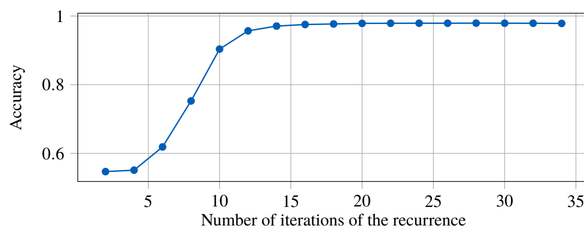


Figure 13: Impact of the number of iterations of Equation (5) on the prediction accuracy of the *Satisfiability* task.

V. Related Work

Motivated by the problems arising from the complexity of manual network operations especially under link failures [15, 16], much progress has been made over the last years towards more automated network operation and verification [2, 17, 18, 19, 3, 20, 21]. Existing network verification tools are typically fed with some model or configuration of the control plane and/or the data plane, and some query. While some tools are specific to a certain protocol, e.g., BGP [22], others are generic [3]. A well-known tool is NetKAT [2] which supports static verification of reachability, loop-freedom or waypoint enforcement, of the network configuration. Other well-known tools include HSA [3] (which is based on a geometric model from the packet headers ignoring protocol-specific meanings), VeriFlow [20] (acting as a layer between the network and an SDN controller), or Anteater [18] (based on a SAT solver).

In contrast to these works, we in this paper focus on MPLS networks, which are in wide use [23]. In particular, we are motivated by a recent line of research which showed that MPLS networks can be verified in polynomial-time, using a connection to prefix rewriting systems and automata theory [5]. The resulting tool, P-Rex [4], relies on a natural query language based on regular expressions which we also adopt in this paper. However, while efficient in theory and much faster than state-of-the-art tools in practice, especially under multiple link failures, P-Rex still requires an hour or more to test complex but relatively small networks.

We in this paper presented a first study of the feasibility of using deep learning to support faster answers to MPLS queries, as well as to synthesize configurations: an emerging topic [24, 25, 15, 26, 27, 28, 29] which to the best of our knowledge however has not yet been studied in the context of MPLS networks so far.

While our methodology is novel in this context, Graph Neural Networks have been around for quite some time [6, 7], and have also been extended to Gated Graph Neural Networks in [11], by using GRU memory units [12]. Message-passing neural networks were introduced in [9], with the goal of unifying various GNN and graph convolutional concepts. Veličković et al. [30] formalized graph attention networks, which enable to learn edge weights of a node neighborhood. Finally, [31] introduced the graph networks (GN) framework, a unified formalization of many concepts applied in GNNs. While existing applications are broad, including chemistry with molecule analysis [8, 9], jet physics and elementary particles [10], prediction of satisfiability of SAT problems [32], or basic logical reasoning tasks and program verification [11], only recently, first applications in networking have emerged, e.g., in the context of network calculus [33, 34], queuing theory [35], protocol generation [36], or the performance evaluation of networks with TCP flows for predicting average flow bandwidth [37].

VI. Conclusion

This paper showed that deep learning can not only be used for a faster verification of the policy-compliance of MPLS configurations, but even has the potential to provide efficient synthesis, automatically re-establishing certain network properties. To achieve this, *DeepMPLS* relies on a novel extension of graph-based neural networks. Our prototype implementation

shows promising results, in terms of runtime and accuracy, in realistic scenarios.

In general, we understand our paper as a first step and believe that our work opens several interesting directions for future work. In particular, we believe that our approach can be refined and optimized further, to provide an even better performance. Furthermore, it will be interesting to investigate the synthesis of full MPLS configurations based on reinforcement learning, or to test and generalize our approach for other configurations, e.g., based on Segment Routing.

In order to facilitate future research in this area and build upon our work, as well as to ensure reproducibility, we made the generated experimental data available online.

ACKNOWLEDGMENTS: The authors would like to thank Jiří Srba for his comments on a early version of this work. We are also grateful to our shepherd, Alex Liu, for his feedback and support. This work was supported by the German-French Academy for the Industry of the Future.

REFERENCES

[1] S. Peter, U. Javed, Q. Zhang, D. Woos, T. Anderson, and A. Krishnamurthy, "One tunnel is (often) enough," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 99–110.
[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *SIGPLAN Not.*, vol. 49, no. 1, Jan. 2014.
[3] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of USENIX NSDI*, 2012.
[4] J. S. Jensen, T. B. Krogh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures," in *Proc. 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.
[5] S. Schmid and J. Srba, "Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks," in *Proc. of IEEE INFOCOM*, 2018.
[6] M. Gori, G. Monfardini, and F. Scarselli, "A New Model for Learning in Graph Domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, ser. IJCNN'05, vol. 2. IEEE, Aug. 2005, pp. 729–734.
[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
[8] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. of NIPS*, 2015.
[9] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. of NIPS*, 2017.
[10] I. Henrion, K. Cranmer, J. Bruna, K. Cho, J. Brehmer, G. Louppe, and G. Rochette, "Neural message passing for jet physics," in *Proc. of the Deep Learning for Physical Sciences Workshop*, 2017.
[11] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks," in *Proc. of ICLR*, 2016.
[12] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. of EMNLP*, 2014.
[13] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 1, pp. 39–47, Mar. 1960.
[15] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations," in *Proc. of ACM SIGCOMM*. ACM, 2016, pp. 328–341.
[14] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.
[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proc. of USENIX OSDI*, 2016.
[17] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for netkat," in *ACM SIGPLAN Notices*, vol. 50 (1), 2015, pp. 343–355.
[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41 (4), 2011, pp. 290–301.
[19] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. ACM, 2017, pp. 155–168.
[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," in *Proc. of USENIX NSDI*, 2013, pp. 15–27.
[21] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. of USENIX NSDI*, 2014, pp. 87–99.
[22] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott, "FSR: Formal analysis and implementation toolkit for safe interdomain routing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1814–1827, 2012.
[23] Y. Vanaubel, P. Mérindol, J.-J. Pansiot, and B. Donnet, "MPLS Under the Microscope: Revealing Actual Transit Path Diversity," in *Proc. ACM IMC*, 2015.
[24] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 261–281.
[25] ——, "NetComplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. of USENIX NSDI*, 2018.
[26] Propane Language. [Online]. Available: https://propane-lang.org/
[27] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Network Configuration Synthesis with Abstract Topologies," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. ACM, pp. 437–451.
[28] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. of USENIX NSDI*, 2018.
[29] ——, "Network-wide configuration synthesis," in *Proc. International Conference on Computer Aided Verification (CAV)*. Springer, 2017.
[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. of ICLR*, 2018.
[31] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018, arxiv:1806.01261.
[32] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT Solver from Single-Bit Supervision," Feb. 2018.
[33] F. Geyer and G. Carle, "The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds," in *Proc. of NetCal*, 2018.
[34] F. Geyer and S. Bondorf, "DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks," in *Proc. IEEE INFOCOM*, 2019.
[35] K. Rusek and P. Cholda, "Message-Passing Neural Networks Learn Little's Law," *IEEE Commun. Lett.*, 2018.
[36] F. Geyer and G. Carle, "Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning," in *Proc. SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks (Big-DAMA)*. ACM, Aug. 2018, pp. 40–45.
[37] F. Geyer, "Performance Evaluation of Network Topologies using Graph-Based Deep Learning," in *Proc. of EAI ValueTools*, 2017.

A.2.4 Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning

This work was published in *Proceedings of the 2018 SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, 2018 [68].

Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning

Fabien Geyer
Technical University of Munich
Garching b. München, Germany
fgeyer@net.in.tum.de

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

ABSTRACT

Automated network control and management has been a long standing target of network protocols. We address in this paper the question of automated protocol design, where distributed networked nodes have to cooperate to achieve a common goal without a priori knowledge on which information to exchange or the network topology. While reinforcement learning has often been proposed for this task, we propose here to apply recent methods from semi-supervised deep neural networks which are focused on graphs. Our main contribution is an approach for applying graph-based deep learning on distributed routing protocols via a novel neural network architecture named Graph-Query Neural Network. We apply our approach to the tasks of shortest path and max-min routing. We evaluate the learned protocols in cold-start and also in case of topology changes. Numerical results show that our approach is able to automatically develop efficient routing protocols for those two use-cases with accuracies larger than 95 %. We also show that specific properties of network protocols, such as resilience to packet loss, can be explicitly included in the learned protocol.

CCS CONCEPTS

• Networks → Network protocol design; **• Computing methodologies → Distributed artificial intelligence**; **Neural networks**;

KEYWORDS

Routing, Graph Neural Network, Deep Learning

ACM Reference Format:
Fabien Geyer and Georg Carle. 2018. Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning. In *Big-DAMA'18: ACM SIGCOMM 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks , August 20, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229607.3229610

1 INTRODUCTION

The current improvements in computational power in packet processing devices and the ability to instrument always more measurements about network performance and behavior have resulted in the emergence of a new paradigm in networking, namely *data-driven networks* and *data-driven protocols* [6, 12]. This new paradigm aims at bringing lessons learned from measurements and data in protocol behavior and design. A concrete application is the concept of *self-driving network* proposed by Feamster and Rexford [6], where network control is tightly coupled with measurements and performance objectives. In this context, machine learning has often been proposed and applied to various tasks such as routing [3, 25, 28], computing resource management [16] or packet scheduling [4].

We propose in this paper to investigate the question of automatic network protocol design using recent methods from semi-supervised deep learning. Our contribution is a novel approach for training a network of independent agents such that they cooperatively exchange information and solve a common goal in a fully distributed manner without central control. We address more specifically the question of distributed routing protocols. From a network protocol perspective, the routing agents should autonomously develop a network protocol akin to RIP or OSPF, *i.e.* exchange topology information and perform local path computations based on the exchanged information. Traditional properties from routing protocols are also considered, namely handling topology changes and packet losses.

Our approach is based on a novel extension of Graph Neural Network (GNN) [10, 21], which we name here *Graph-Query Neural Network* (GQNN). GNNs are neural network architectures able to process graphs as input using the concept of message passing between nodes in the graph. We evaluate our approach on various topologies from real networks [13] and show that our approach leads to the creation of communication protocols able to exchange data about topology information as well as topology changes. Using two different path calculation strategies – namely shortest path routing and max-min fair routing – we show that various routing objectives can be achieved using the same neural network architecture. The results of our approach are also compared against bounds on information propagation in topologies and we show that the routing protocols are efficient in term of number of iterations necessary to reach convergence. We also demonstrate that resilience to packet loss can be explicitly trained for.

This work is structured as follows. We describe in Sections 2 and 3 our modeling approach and the neural network architecture, with an introduction on Graph Neural Networks and Graph Gated Neural Networks, followed by the application of those concepts to network topologies and distributed routing protocols. We numerically evaluate our approach in Section 4 with the evaluation on real network topologies. In Section 5, we present similar research studies. Finally, Section 6 concludes our work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Big-DAMA'18, August 20, 2018, Budapest, Hungary
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5904-7/18/08…$15.00
https://doi.org/10.1145/3229607.3229610

2 NEURAL NETWORKS FOR GRAPHS

The main intuition behind our approach is to map network topologies to graphs, with nodes representing routers and additional nodes for each physical interface of the router. Those graph representations are then used as input for a neural network architecture able to process general graphs. The transformation from a network topology to its graph representation is presented in Section 3.

In this section, we detail the neural network architecture used for learning on graphs, namely the family of architectures based on Graph Neural Networks [10, 21], and introduce a new extension of this architecture.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. Let $\mathbf{i}_v$ and $\mathbf{o}_v$ represent respectively the input features and target values of node v. The concept behind Graph Neural Networks is called *message passing*, where hidden representations of nodes are based on the hidden representations of their neighboring nodes. Those hidden representations are propagated through the graph using multiple iterations until a fixed point is found. The final hidden representation is then used for predicting properties about nodes. This concept can be expressed as:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}_u^{(t-1)} \,\middle|\, u \in \textsc{Nbr}(v)\right\}\right) \tag{1}$$

$$\mathbf{o}_v = g\left(\mathbf{h}_v^{(t\to\infty)}\right) \tag{2}$$

$$\mathbf{h}_v^{(t=0)} = init\,(\mathbf{i}_v) \tag{3}$$

with $\mathbf{h}_v^{(t)}$ representing the hidden representation of node v at time t, $f(\cdot)$ a function which aggregates the different hidden representations, $\textsc{Nbr}(v)$ the set of neighboring nodes of v, $g(\cdot)$ a function for transforming the final hidden representation to the target values, and $init(\cdot)$ a function for initializing the hidden representations based on the input features.

The concrete implementations of the $f(\cdot)$ and $g(\cdot)$ functions are feed-forward neural networks, with the special case that $f(\cdot)$ in Equation (1) is the sum of per-edge terms (as recommended by [21]) such that:

$$\mathbf{h}_v^{(t)} = f\left(\left\{\mathbf{h}_{\textsc{Nbr}(v)}^{(t-1)}\right\}\right) = \sum_{u \in \textsc{Nbr}(v)} f^*\left(\mathbf{h}_u^{(t-1)}\right) \tag{4}$$

with $f^*(\cdot)$ a feed-forward neural network. For $init(\cdot)$, the initial node features are used and zero-padded to fit the dimensions of the hidden representations.

In [10, 21], training the neural network architecture, namely the parameters of $f(\cdot)$, $g(\cdot)$ and $h(\cdot)$, is done via the Almeida-Pineda algorithm [2, 20] which works by running the propagation of the hidden representation to convergence, and then computing gradients based upon the converged solution.

2.1 Extensions of Graph Neural Networks

Various extensions of GNNs have been proposed in the literature in recent years in order to improve accuracy and applicability. Those extensions build on the principle of message passing with more recent development from deep learning. We give here an overview over the extensions which were used for the final architecture used in this paper. For easier notation, we define $\mathbf{H}^{(t)}$ as the vector of all hidden representations at iteration t: $\left[\mathbf{h}_1^{(t)} \cdots \mathbf{h}_{|\mathcal{V}|}^{(t)}\right]$.

2.1.1 Gated Graph Neural Network. In order to improve the training of Graph Neural Networks, Li et al. proposed Gated Graph Neural Networks (GG-NNs) in [14]. This extension implements the function $f(\cdot)$ using a memory unit called Gated Recurrent Unit (GRU) [5] and unrolls Equation (1) for a fixed number of iterations.

Formally, the propagation of the hidden representations among neighboring nodes for one time-step is formulated as:

$$\mathbf{x}^{(t)} = \mathbf{H}^{(t-1)}\mathbf{A} + \mathbf{b}_a \tag{5}$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_z x^{(t)} + \mathbf{U}_z\mathbf{H}^{(t-1)} + \mathbf{b}_z\right) \tag{6}$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_r x^{(t)} + \mathbf{U}_r\mathbf{H}^{(t-1)} + \mathbf{b}_r\right) \tag{7}$$

$$\widetilde{\mathbf{H}}^{(t)} = \tanh\left(\mathbf{W}x^{(t)} + \mathbf{U}\left(\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)}\right) + \mathbf{b}\right) \tag{8}$$

$$\mathbf{H}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \odot \mathbf{H}^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{H}}^{(t)} \tag{9}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function and $\odot$ is the element-wise matrix multiplication. $\{\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}\}$ and $\{\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}\}$ are trainable weight matrices, and $\{\mathbf{b}_a, \mathbf{b}_r, \mathbf{b}_z, \mathbf{b}\}$ are trainable bias vectors. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the graph adjacency matrix, determining how nodes in the graph $\mathcal{G}$ communicate with each other.

2.1.2 Edge attention. A recent advance in neural networks has been the concept of *attention*, which provides the ability to a neural network to focus on a subset of its inputs. This mechanism has been used in a variety of applications such as computer vision or natural language processing (eg. [26]). For the scope of GNNs, we introduce here so-called *edge attention*, namely we wish to give the ability to each node to focus only on a subset of its neighborhood. Formally, let $a_{(v,u)}^{(t)} \in [0, 1]$ be the attention between node v and u. Equation (4) is then extended as:

$$\mathbf{h}_v^{(t)} = \sum_{u \in \textsc{Nbr}(v)} a_{(v,u)}^{(t)} \cdot f^*\left(\mathbf{h}_u^{(t-1)}\right) \tag{10}$$

$$\mathbf{a}_{(v,u)}^{(t)} = f_A\left(\mathbf{h}_v^{(t-1)}, \mathbf{h}_u^{(t-1)}\right) \tag{11}$$

To make the computation of $a_{(v,u)}^{(t)}$ for all edges more efficient, we use the modified adjacency matrix $\widetilde{A}^{(t)}$ defined as:

$$\widetilde{A}^{(t)} = A \odot \sigma\left(f_{A_1}\left(\mathbf{H}^{(t)}\right)^T \odot f_{A_2}\left(\mathbf{H}^{(t)}\right)\right) \tag{12}$$

with $f_{A_1}(\cdot)$ and $f_{A_2}(\cdot)$ feed-forward neural networks. Similar concepts of edge attention have already been proposed in the literature with various implementations (eg. [11, 27]).

3 APPLICATION TO ROUTING

We describe in this section the application of the graph-based deep learning architectures presented in Section 2 to the task of distributed routing protocols. We are interested in training neural networks on two important aspects of those network protocols. The first one is the network protocol itself, namely how to distribute topology information among different nodes, and the second one is how to compute routes given a topology and link weights.

3.1 Graph representation

For the first aspect, we wish to train routing agents such that they autonomously exchange data about a given topology without explicitly specifying which information about the topology to transmit. Based on this exchanged information, the routing agent can populate routing tables. In comparison to traditional distributed routing protocols, we essentially wish to train neural networks to transmit information akin to link-state updates (as used in OSPF for example) or router distances (as used in RIP for example). As for standard routing protocols, the learned protocol should also deal with changes in the topology (i.e. link failure, node addition).

The main intuition behind the input feature modeling is to use the topology as input graph $\mathcal{G}$, with additional nodes representing the network interfaces as illustrated in Figures 1a and 1b. In order to enforce communication between nodes according to the physical network topology, no additional edge is added to the graph. As for traditional routing protocols, each router in the topology is assigned an integer identifier, noted R_i. Nodes representing routers in the graph use this identifier encoded as a one-hot vector for their initial representation $\mathbf{i}_v$. Nodes representing interfaces use a weight parameter (eg. based on the link bandwidth) for $\mathbf{i}_v$.

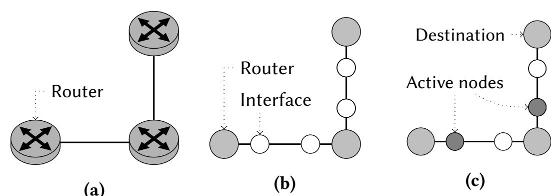


Figure 1: (a) Example network topology. (b) Its associated graph used for training. (c) The output feature of the interfaces according to a selected destination.

3.2 Graph Query Neural Networks

We are interested in this section in the local computation of the routing table based on the topology information which was distributed by the different nodes in the graph. Given a destination router identifier R_d, each router must locally decide which output interface should be used. Based on the graph representation from the previous section and a given algorithm for path calculation, this is modeled by labeling the interfaces with $\mathbf{o}_i = [1]$ if they are used for transmitting packets to router R_d, and $[0]$ otherwise, as illustrated in Figure 1c.

In order to build a routing protocol with local path computation, we introduce here a new extension to GNNs, called *Graph Query Neural Network* (GQNN). The neural network architecture is illustrated in Figure 2. The hidden node representations $\mathbf{h}_v^{(t)}$ correspond to the messages which are transfered between nodes. Once the message passing is finished, each node in the graph has a local representation of the network topology $\mathbf{h}_v^{(T)}$. For determining which interface to use for a given destination router R_d, each router applies the following procedure on each of its interface nodes:

$$\mathbf{q}_d = Q(R_d) \qquad \text{query vector computation} \qquad (13)$$

$$\mathbf{o}_v = g\left(\mathbf{q}_d \odot \mathbf{h}_v^{(T)}\right) \qquad \text{output label as in Equation (2)} \qquad (14)$$

with $Q(\cdot)$ and $g(\cdot)$ feed-forward neural networks.

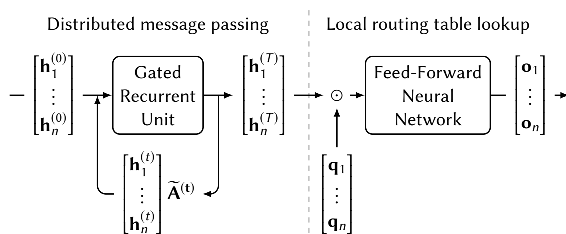


Figure 2: Overview of the Graph Query Neural Network architecture used in this paper

3.3 Learned routing strategies

For the scope of this paper, we evaluate two algorithms for route calculation: *shortest path* and *max-min routing*.

In the case of shortest path routing, the neural network is trained against path calculations based on Dijkstra's algorithm, where each link is associated with a weight. In case multiple shortest paths are available, we need to discriminate between them in order to have stable routing strategies for easier training of the neural network. This is performed by using the router identifiers as discriminant between paths.

In the case of max-min fair routing [18], we aim at maximizing the minimum allocated bandwidth between all possible source-destination pairs in the network. Such routing strategy should lead to network topologies with less link overload than shortest path routing. For our evaluations, we assign an equal demand for all possible source-destination pairs. The route computation is done using linear programing. As for shortest path routing, we also give priority to paths which minimize the identifiers of the traversed routers. This is performed by defining multiple objectives and solving them in a hierarchical way.

3.4 Packet losses and topology changes

An important requirement of routing protocols is the ability to cope with packet losses and dynamic topology changes. In the case of packet losses, various strategies have been used in existing routing protocols: either leverage transport protocols (*e.g.* BGP over TCP), or design an own transport layer (*e.g.* OSPF). For this paper, we are interested in designing network protocols which do not leverage other transport protocol functionalities.

In order to train the neural network to handle packets loss, the adjacency matrix $\mathbf{A}$ is randomly modified during training such that some edges are temporarily disabled according to a Bernoulli distribution with parameter p. We implemented this by using a dropout layer [23] in the neural network architecture without the traditional normalization factor used in standard dropout:

$$\hat{\mathbf{A}}^{(t)} = \mathbf{r}(t) \odot \mathbf{A} \quad \text{with } \mathbf{r}(t) \sim \mathit{Bernoulli}(p),\ \forall t \in [0, T] \qquad (15)$$

Since routing protocols are designed to run continuously and handle topology changes, we define here two phases of the protocol: *cold-start* when the routing protocol is first initialized on the active routers, and *warm-start* when a node fails or a new node joins a network where the routing protocol already ran for some iterations. More formally, we define graphs pairs $\{\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1), \mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)\}$ such that some routers are added or removed between $\mathcal{G}_1$ and $\mathcal{G}_2$. The neural network is first trained on $\mathcal{G}_1$, and the final hidden representations from this first phase are then reused as initial hidden representations for the second training phase on $\mathcal{G}_2$ for the nodes which did not change between $\mathcal{G}_1$ and $\mathcal{G}_2$:

$$\forall v \in \{\mathcal{V}_1 \cap \mathcal{V}_2\}, \quad \mathbf{h}_{v,\mathcal{G}_1}^{(t=T)} = \mathbf{h}_{v,\mathcal{G}_2}^{(t=0)} \tag{16}$$

3.5 Implementation in routers

Regarding implementation of the resulting network protocol in real routers, each router has an internal subgraph, with one central node representing the router connected to multiple nodes representing its interfaces. Each router then periodically broadcasts its hidden interface representations to its neighboring routers, and recomputes its routing table based on the received messages.

4 NUMERICAL EVALUATION

We evaluate in this section the approach presented in Sections 2 and 3 on real network topologies from the *Internet Topology Zoo* [13], which is a collection of topologies from Internet providers around the world. We selected topologies such that the number of nodes is limited to 20 and the maximum hop count between any two nodes in the network is less than 10.

In order to generate more topologies for training our neural network, each topology is modified by either randomly adding a router and connecting it randomly to other routers, or randomly deleting one router in the topology. Router identifiers are randomly assigned to the routers as described in Section 3. Random router failure or addition are generated as described in Section 3. This resulted in a dataset with 40 000 graphs in total.

The two routing algorithms presented in Section 3.3 are then applied on each generated topology to build the datasets used for this evaluation.

4.1 Implementation

The GG-NN architecture presented in Sections 2 and 3.2 was implemented using Tensorflow [1] and trained using Nvidia GPUs. Additional dropout layers [23] were added according to standard practices for neural network and recurrent neural networks [22] in order to avoid over-fitting. Hidden node representations were chosen with a size of 160. The same parameters were used for training the neural network for both routing use-cases.

4.2 Accuracy

We evaluate in Figure 3 the accuracy of the computed routes according to the two use-cases and routing phases. For a given topology, we define the accuracy as 1 if the route for a given destination is correct for all routers in the topology, and 0 otherwise. The learned protocol is better able to predict shortest path routing, where a perfect accuracy is reached for than 50 % of the evaluated topologies.

In average, accuracies of 98 %, respectively 95 %, could be reached for shortest path routing, respectively min-max routing.

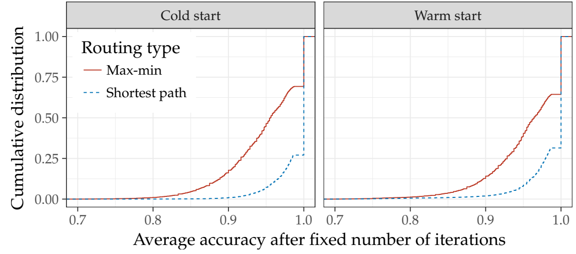


Figure 3: Overview over the accuracy of the predicted routes

4.3 Convergence time

In order to assess the convergence time of the developed protocols, we first evaluate the accuracy of the routing at different iterations of the fixed point evaluation presented in Equation (1) in cold-start and warm-start phases. The numerical results are presented in Figure 4. In case of topology changes (ie. warm start), better accuracies are reached faster as routes only need partial reconfiguration. This shows that the protocol is indeed able to efficiently cope with and react to topology changes.

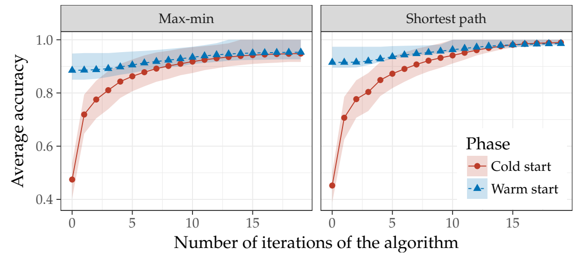


Figure 4: Accuracy according to the iterations of the protocols. Areas indicate the 25 and 75 percentile.

We then evaluate the developed protocols against minimum bounds of the number of iterations required to achieve convergence in the computed routes. According to the message passing principles described in Section 2, each node broadcasts learned information at each iteration of the routing protocol to its neighbors. In order to achieve a correct computation of the routes, each node needs to have received at least some piece of information from all the other nodes in the topology. Hence, the minimum number of iterations corresponds to the diameter d of the graph of the network topology. We call this minimum bound the *one-way bound*. In case any pair of nodes in the topology needs to have exchanged information in both directions, the minimum number of iterations needed is $2d$, called here *both-way bound*.

We define T_C as the iteration when convergence occurs, i.e. the time when the computed routes for each node in the topology are stable. Figure 5 compares T_C against the two bounds previously

defined, namely for each graph we compute $T_C - d$ and $T_C - 2d$. Negative values indicate that the learned protocol converged faster than the theoretical bound.

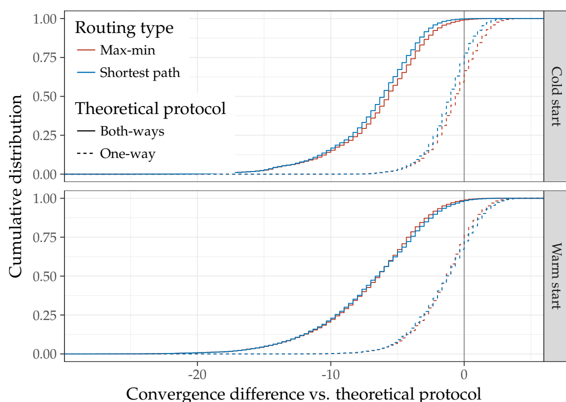


Figure 5: Evaluation of cold-start and warm-start convergence time of the learned protocols against theoretical bounds. Negative values indicate fast convergence.

4.4 Resilience to packet loss

We described in Section 3.4 that a crucial property of routing protocol is resilience to packets loss. Figure 6 illustrates the accuracy of the protocols in case of different packet loss probabilities in the network. We also compare in Figure 6 two different variants for training the neural network, namely explicit or unspecific training for handling packet loss. Both variants of the protocol are run for the same number of iterations. Explicit training for packet loss was done by using a dropout layer as described in Equation (15).

We notice a clear difference between the two training variants. By explicitly training for packet loss, the learned protocol is able to reach better accuracy in case of packet loss.

4.5 Visualization

In order to better understand the working of the generated routing protocol, we propose in this section to visualize information propagation in a topology. Figure 7 illustrates the accuracy of a given route on a small topology at different iterations of the protocol. Such visualization can be used to determine protocol convergence for the different interfaces in the network.

5 RELATED WORK

The question of distributed routing protocols based on machine learning has already attracted various researchers. Early work on this topic include *Q-Routing* from Boyan and Littman [3], *COllective INtelligence* (COIN) from Wolpert et al. [28], or *distributed Gradient Ascent Policy Search* (GAPS) from Peshkin and Savova [19]. Their general approach is to use multi-agent reinforcement learning in combination with a network-wide utility function. More recently, Valadarsky et al. [25] also proposed to use reinforcement learning, with the goal of using past traffic matrices in order to guide route calculations. Compared to those works, we use here semi-supervised

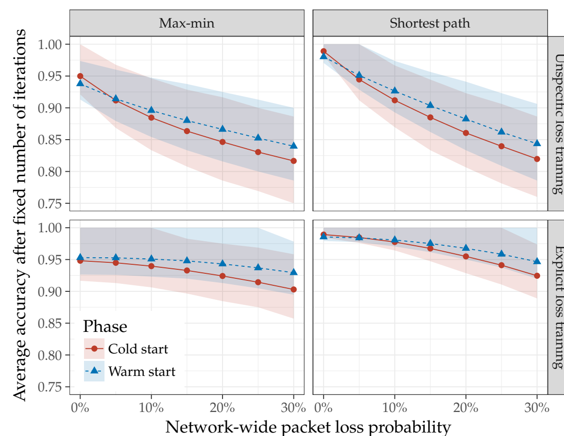


Figure 6: Accuracy of the protocols in case of packet loss in the network with explicit or unspecific training for packet loss. Areas indicate the 25 and 75 percentile.

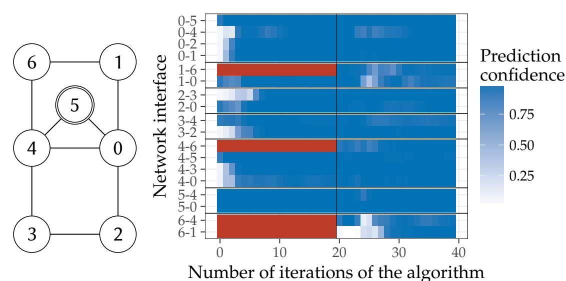


Figure 7: Visualization of the protocol evolution on a small network topology. Each network interface is queried for node 5. Node 6 is first offline and started at iteration 20.

learning in order to more easily specify the routing policy which is expected. Previous work also often predetermined or constrained the specification and format of the communication, whereas our approach leaves the content or format of the exchanged information as a parameter to be learned. Our work also evaluates key aspects of routing protocols, namely resilience against packet loss and inclusion of network dynamics.

A supervised learning approach was recently proposed by Mao et al. [15] using Supervised Deep Belief Architectures, with a focus on speed of route computation. Compared to their approach, our method can be applied to a wider range of network topologies since it is independent of the underlying structure of the topology.

The challenge of training agents to communicate and realize a common goal has attracted work in other domains. Foerster et al. [7] applied *Deep Distributed Recurrent Q-Networks* (DDRQN) for solving logic riddles. Sukhbaatar et al. [24] proposed a deep neural network architecture called *CommNet* for developing communication between agents on the task of multi-turn games, traffic junction or

logic riddles. In both approaches, no constraint on communication structure is enforced as a broadcast channel is used.

Neural networks for graphs have recently attracted a larger interest, and are generally based on the concept of message passing presented in Section 2. In the context of communication networks, they have successfully been applied to performance evaluation of TCP flows in [8]. The model presented here is based on [8] with novel extensions for edge attention, query as presented in Section 3.2, and support training for topology changes. They have also been used in a variety of other domains such as basic logical reasoning tasks and program verification [14], semantic role labeling in natural language processing [17], prediction of chemical properties of molecules [9]. To the best of our knowledge, this is the first work applying GNNs to distributed routing protocols.

6 CONCLUSION

We contributed in this paper a novel approach for automatic network protocol design using graph-based deep learning. Our method is based on an extension of Graph Neural Networks called Graph Query Neural Network and a mapping from network topologies to graphs with special nodes representing network interfaces.

We applied our approach to distributed routing protocols, where routing nodes need to exchange information about the network topology in order to reach efficient routes without specifying which information to exchange. Shortest path and max-min routing were evaluated as routing strategies. In our numerical evaluation, we showed that our approach is able to reach good accuracies. We illustrated that specific properties of network protocols such as resilience to packet loss can be explicitly included in the learned protocols by training the neural network with appropriate dropout.

As our approach is not specific to routing protocols, future work may include evaluations and extensions of our approach to other network protocols and applications.

Acknowledgments This work was supported by the German Federal Ministry of Education and Research (grant 16KIS0538, project DecADe), by the German-French Academy for the Industry of the Future, and the High-Performance Center for Secure Networked Systems.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Luis B. Almeida. 1990. Artificial Neural Networks. IEEE Press, Piscataway, NJ, USA, Chapter A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment, 102–111.

[3] Justin A. Boyan and Michael L. Littman. 1994. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector (Eds.). Morgan-Kaufmann, 671–678.

[4] Timothy X. Brown. 2002. Switch Packet Arbitration via Queue-Learning. In *Advances in Neural Information Processing Systems 14*, T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.). MIT Press, 1337–1344.

[5] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. (June 2014). arXiv:1406.1078

[6] Nick Feamster and Jennifer Rexford. 2017. Why (and How) Networks Should Run Themselves. (Oct. 2017). arXiv:cs.NI/1710.11583v1

[7] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. 2016. Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks. (Feb. 2016). arXiv:cs.AI/1602.02672v1

[8] Fabien Geyer. 2017. Performance Evaluation of Network Topologies using Graph-Based Deep Learning. In *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2017)*. https://doi.org/10.1145/3150928.3150941

[9] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1263–1272.

[10] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A New Model for Learning in Graph Domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks (IJCNN'05)*, Vol. 2. IEEE, 729–734. https://doi.org/10.1109/IJCNN.2005.1555942

[11] Yedid Hoshen. 2017. VAIN: Attentional Multi-agent Predictive Modeling. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 2698–2708.

[12] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. 2017. Unleashing the Potential of Data-Driven Networking. In *Proceedings of 9th International Conference on COMmunication Systems & NETworkS (COMSNET)*.

[13] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE J. Sel. Areas Commun.* 29, 9 (Oct. 2011), 1765–1775. https://doi.org/10.1109/JSAC.2011.111002

[14] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'2016)*.

[15] Bomin Mao, Zubair Md. Fadlullah, Fengxiao Tang, Nei Kato, Osamu Akashi, Takeru Inoue, and Kimihiro Mizutani. 2017. Routing or Computing? The Paradigm Shift Towards Intelligent Computer Network Packet Transmission Based on Deep Learning. *IEEE Trans. Comput.* 66, 11 (Nov. 2017), 1946–1960. https://doi.org/10.1109/TC.2017.2709742

[16] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. 50–56. https://doi.org/10.1145/3005745.3005750

[17] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. (March 2017). arXiv:cs.CL/1703.04826v4

[18] Dritan Nace and Michał Pióro. 2008. Max-Min Fairness and Its Applications to Routing and Load-Balancing in Communication Networks: A Tutorial. *IEEE Commun. Surveys Tuts.* 10, 4 (2008), 5–17. https://doi.org/10.1109/SURV.2008.080403

[19] Leonid Peshkin and Virginia Savova. 2002. Reinforcement Learning for Adaptive Routing. In *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1825–1830. https://doi.org/10.1109/IJCNN.2002.1007796

[20] Fernando J. Pineda. 1987. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.* 59 (Nov. 1987), 2229–2232. Issue 19. https://doi.org/10.1103/PhysRevLett.59.2229

[21] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Netw.* 20, 1 (Jan. 2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[22] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. 2016. Recurrent Dropout without Memory Loss. (March 2016). arXiv:1603.05118

[23] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (Jan. 2014), 1929–1958.

[24] Sainbayar Sukhbaatar, Rob Fergus, et al. 2016. Learning Multiagent Communication with Backpropagation. In *Advances in Neural Information Processing Systems*. 2244–2252.

[25] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. 2017. Learning to Route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. ACM, New York, NY, USA, 185–191. https://doi.org/10.1145/3152434.3152441

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 6000–6010.

[27] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. (Oct. 2017). arXiv:stat.ML/1710.10903v1

[28] David Wolpert, Kagan Tumer, and Jeremy Frank. 1999. Using Collective Intelligence to Route Internet Traffic. In *Advances in Neural Information Processing Systems 11*, M. J. Kearns, S. A. Solla, and D. A. Cohn (Eds.). MIT Press, 952–960.

A.3 Hardware and software for efficient packet processing

A.3.1 Cryptographic Hashing in P4 Data Planes

This work was published in *Proceedings of the 2nd P4 Workshop in Europe*, 2019 [157].

2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)

Cryptographic Hashing in P4 Data Planes

Dominik Scholz*, Andreas Oeldemann†, Fabien Geyer*, Sebastian Gallenmüller*,
Henning Stubbe*, Thomas Wild†, Andreas Herkersdorf†, Georg Carle*

Technical University of Munich, Germany
*{scholz,fgeyer,gallenmu,stubbe,carle}@net.in.tum.de
†{andreas.oeldemann,thomas.wild,herkersdorf}@tum.de

***Abstract*—P4 introduces a standardized, universal way for data plane programming. Secure and resilient communication typically involves the processing of payload data and specialized cryptographic hash functions. We observe that current P4 targets lack the support for both. Therefore, applications and protocols, which require message authentication codes or hashing structures that are resilient against attacks such as denial-of-service, cannot be implemented.**

To enable authentication and resilience, we make the case for extending P4 targets with cryptographic hash functions. We propose an extension of the P4 Portable Switch Architecture for cryptographic hashes and discuss our prototype implementations for three different P4 target platforms: CPU, NPU, and FPGA. To assess the practical applicability, we conduct a performance evaluation and analyze the resource consumption. Our prototype implementations show that cryptographic hashing can be integrated efficiently. We cannot identify a single hash function delivering satisfying performance on all investigated platforms. Therefore, we recommend a set of hash functions to optimize target-specific performance.

***Index Terms*—Hash function, Data Plane Programming, Performance Evaluation, P4**

I. Introduction

The rise of paradigms like P4 [1] for programming high-speed packet processing platforms has enabled a shift of networking applications to the data plane. Examples of such applications include heavy-hitter detection [2], [3] and in-network caching for distributed services [4]. Looking at implementations of those applications, hash-based data structures like hash tables, bloom filters, or count–min sketches often serve as a basis for efficiently tracking flows. Currently, P4 only supports few algorithms for hash functions, based on cyclic redundancy check (CRC) or checksum calculations commonly used in network protocols (e.g., IP, TCP checksums), which can operate on the header fields of a packet.

To address more secure and advanced applications in the data plane, a wider set of hash functions with cryptographic properties may be beneficial. Two classes of applications can benefit: First, resilience to hash collisions can be improved for hash-based data structures. High susceptibility to hash collisions can create attack vectors, leading to poor resource usage or denial of services [5]. Second, integrity protection, which is typically implemented using hash-based message authentication codes (HMAC) for instance for digital signatures or challenge-response protocols. These are essential for secure communication not only on the Internet but also in industrial networks.

We argue for the benefits of including cryptographic hash functions in P4 platforms. We present our prototype implementations for three different P4 targets: the t4p4s software platform, the Netronome Agilio NFP-4000 Smart NIC, and the NetFPGA SUME. Using measurements we discuss the impacts on performance and resource consumption of cryptographic hash implementations for these devices.

The rest of this paper is organized as follows: First, we review related work in Section II. We argue for the inclusion of cryptographic hash functions in programmable packet processing platforms in Section III. In Section IV we discuss our approaches to extend three different P4 targets with the functionality to calculate cryptographic hashes over packet data. We conduct an evaluation of our prototype implementations focusing on performance metrics as well as resource consumption in Section V. Section VII concludes our work.

II. Related work

Various work already evaluated the suitability of hash algorithms for network packets. Molina et al. [6] and Henke et al. [7] evaluate several different functions, with a focus on packet sampling. Both works highlight that CRC32 is not recommended due to its linear dependency between hash input and hash value, making it vulnerable to bias and security attacks. They recommend BOB [8] as hash algorithm in non-adversarial scenarios due to its performance and avalanche properties. Regarding hardware implementation of non-cryptographic hash algorithms for networking applications, Hua et al. [9] evaluated 18 different functions. They propose a family of hash functions achieving good properties in terms of hashing at a reduced cost regarding hardware footprint and cost per cycle.

Use of hash functions for networking applications implemented in the P4 data plane can be found in various work. Ghasemi et al. [10] investigate performance diagnostic of TCP with *Dapper* using standard 5-tuple hashes. Zaoxing et al. [2] propose *UnivMon* for network flow monitoring based on a sketch data structure where multiple pairwise-independent hash functions are used. Cidon et al. [11] propose *AppSwitch*, a cache for key-value storage using hashes of keys. Sivaraman et al. [3] introduce *HashPipe*, a heavy-hitter detection using a pipeline of hash tables, which retain counters

978-1-7281-4387-3/19/$31.00 © 2019 IEEE

for heavy flows while being memory efficient. Finally, Kucera et al. [12] also address heavy-hitter detection using *Elastic Trie*, a novel trie-based data structure. The mentioned works either do not detail the hash algorithm used, or make use of CRC32 as hash function, making them potentially vulnerable to security attacks. Only Ghasemi et al. [10] explicitly describe the strategy used to deal with hash collisions. They use a hash chaining technique combining the hashed value and the TCP sequence numbers.

The IEEE 802.1AE (MACsec) standard provides data confidentiality and integrity through a security tag and a message authentication code (MAC) on the data link layer. These properties are especially interesting for industrial use cases, including automotive [13], [14] and aeronautical applications [15]. Hauser et al. [16] propose P4-MACsec for the automation of MACsec deployment by shifting the MACsec implementation entirely to the data plane of P4 targets. They implement prototypes for the BMv2 model and the NetFPGA, but the solution for the latter was not feasible due to the same problems regarding externs we encountered (see Section IV-C). Hauser et al. [17] propose a similar approach for IPsec, but their prototype implementation for the NetFPGA has the same restrictions. For the ASIC prototype, all cryptographic processing is performed by a CPU-based controller, as the ASIC neither offers cryptographic algorithms nor adding them as externs. This limits performance and functionality [17].

III. Motivation

Hash functions play a key role in various network applications being fundamental for modern network communication. As more and more functionality is being moved to programmable data planes, supporting hash functions with strong cryptographic properties will be a key enabler for various networking use cases.

A. Working with hashes in P4

The Portable Switch Architecture of $P4_{16}$ supports five different functions, which may serve as hash functions: four variants of CRC and the 16 bit one's complement used for IP, TCP, and UDP checksum calculation. While these may serve as a good hash function in networking applications [18], they do not provide cryptographic properties.

In $P4_{16}$, hash algorithms can be accessed via standard function calls of external libraries, so-called externs. For instance, the P4 switch model `v1model.p4` external library offers the generic `hash` function. Its parameters include the hash algorithm to use, as well as a list of parsed header or metadata fields to be used as input. The P4 target platform may support additional hashing algorithms as externs.

While P4 does not directly offer primitives for working with data structures such as hash tables or Bloom filters, P4 primitives can be used in combination with P4 registers to emulate those data structures.

B. Applications with security properties

Various attacks have been proposed on poorly implemented hash-based data structures. For instance, hash tables can degenerate to linked lists with maliciously chosen input, leading to high CPU usage in network security monitors [19]. It is therefore recommended to either use cryptographically-strong random number generators or keyed pseudo-random functions instead of CRC [19], [20].

Due to the use of relatively small messages in packet processing, the choice of a hash function for efficient processing is not straightforward. While the SHA-2 family of hash functions is a strong candidate regarding cryptographic features and security, these functions were not designed with good performance for small inputs. Popular candidates are the SipHash family of hash functions used in various programming languages and software [21], or the BLAKE2 family [22], both designed for good performance for small inputs.

Cryptographic hash algorithms are found in various network protocols with different uses [23]. Extending P4 and its hardware platforms with cryptographic algorithms enables offloading of secure applications to the data plane. A use case of interest are MACs, where packet content is checked for data integrity and authenticity. Hash-based MACs (HMACs) are often used for this and can be found in various protocols such as IPsec, TLS or IEEE 802.1AE (MACsec).

Digital signatures or challenge-response protocols, using token or cookie mechanisms, are used to either prove the possession of an authentication token or to encode state that is being exchanged. One such example are TCP SYN cookies [24], which are calculated for each incoming TCP SYN packet during an ongoing attack and, therefore, have to be efficiently generated and verified.

IV. Hashing Extern Implementation

We have extended three different P4 target platforms with externs calculating cryptographic hashes. Each platform has its own way how P4 externs can be added.

A. CPU: t4p4s

t4p4s [25] is a P4 compiler, which generates platform-independent C code. Target-specific code can be linked with additional libraries, in our case the Dataplane Development Kit (DPDK version 17.08). This allows the P4 program to be executed in user space on a CPU-based software system. We use the name t4p4s synonymously for both the $P4_{14}$ compiler and the DPDK-based P4 target.

As t4p4s only supports the TCP/IP checksum calculation as hash algorithm, we extended it with an SSE4.2-accelerated non-cryptographic *CRC32* function. Furthermore, we added the following open-source implementations of (pseudo-) cryptographic hash functions as P4 externs: the original version of *SipHash-2-4* [21]; *Poly1305-AES* [26] based on [27]; the original verion of *BLAKE2b* [22]; and *HMAC-SHA256* and *HMAC-SHA512* based on OpenSSl (v1.1.0). The output length in bit of each of the hash functions is listed in Table I. We

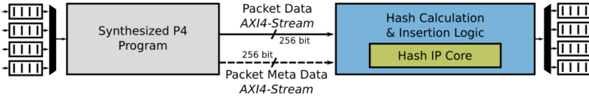


Figure 1: Integration of hash calculation and insertion

refer to related work regarding their cryptographic properties and cryptanalysis.

B. NPU: NFP-4000 SmartNIC

The 10G Netronome Flow Processor (NFP)-4000 Agilio SmartNIC [28] is a Network Processing Unit (NPU) that relies on a 32 bit many-core architecture with up to 60 freely programmable flow processing cores. A P4 compiler is offered by Netronome, which compiles P4 code for the NPU (SDK v6.0.4). While none of the supported hash functions has cryptographic properties, the SmartNIC allows implementing P4 externs in Micro-C, a variation of C used to program the processing cores. Externs are inlined into the compiled P4 program. In addition to the existing P4 hashes for CRC32 and Checksum, we have implemented the SipHash-2-4 function in Micro-C, calculating a hash for the payload of the Ethernet frame. The NFP-4000 features a hardware crypto security accelerator supporting SHA1 and SHA2, however, the accelerator was not available on our NPU, therefore we opted for the CPU-optimized SipHash instead.

C. FPGA: NetFPGA SUME

P4->NetFPGA [29] provides an open-source hardware design for the NetFPGA SUME board, which instantiates $P4_{16}$ programs compiled via Xilinx SDNet (we used version 2018.1). We selected the open-source RTL implementations of a *SipHash-2-4*[1] (64 bit output) and a *SHA3-512*[2] (512 bit output) IP core for integration into our prototype design.

Integrating the hash IP cores seamlessly as P4 externs via interfaces defined by the P4->NetFPGA implementation is not possible. The current design does not implement a streaming interface for extern data in- and output, where data is fragmented into multiple subsequent words. In- and output data is passed among the P4 program and externs as a single data word via a fixed number of parallel wires, requiring thousands of wires for maximum-sized Ethernet frames. We found that the current version of the SDNet compiler is only able to handle input widths of up to approx. 600 B. However, even for an input width of 64 B we were unable to obtain timing closure due to resource congestion.

As an alternative, we have changed the P4 switch model of the P4->NetFPGA design by integrating the hash calculation in the egress path after the synthesized P4 program (Figure 1). Per-packet metadata written by the P4 program instructs the hash module whether and where to insert the hash. As hashes are calculated after packets traverse the P4 program, packet modifications or forwarding decisions relying on hash calculations cannot be implemented in P4. Relocating the hash IP core into the ingress path would allow hashes to be passed to the P4 program via metadata. Another alternative is to further enhance the P4 switch model of the P4->NetFPGA by placing a second P4 pipeline after the hashing module.

Finally, our implementation would benefit from a traffic manager to selectively steer traffic around the IP core to avoid blocking of packets, which do not require hash calculation.

[1]SipHash IP Core: https://github.com/secworks/siphash
[2]SHA3-512 (KECCAK) IP Core: https://github.com/freecores/sha3

D. Limitations

Our extern implementations for the NFP-4000 and NetFPGA SUME do not use key material or an HMAC scheme required to generate a message authentication code, but only calculate a single cryptographic hash. This is done for simplicity and to focus on evaluating the performance of the basic cryptographic operation, which could be applied for use cases other than HMAC calculations. However, this functionality could be added, for instance by providing the key material as part of P4 metadata on a per-packet basis.

V. Prototype Evaluation

Our measurement setup consists of two servers connected via a 10 Gbit/s Ethernet link. One server acts as a load generator and sends packets to the device under test (DuT), which runs an L2 forwarding P4 program that additionally calculates hashes based on the complete Ethernet frames. The server acting as the DuT is equipped with an Intel Xeon CPU E5-2620 v3 (Broadwell) at 2.40 GHz and either an Intel X540 network card, Netronome NFP-4000 SmartNIC, or the NetFPGA SUME. For measurements performed for the CPU target, all traffic is pinned to one CPU core.

A. Metrics for hash functions

Several metrics depending on the requirements of the application and capabilities of the (hardware) platform are of relevance when choosing a hash function. In high-performance applications, the performance of the hash function in terms of latency and processing time (e.g. clock cycles) is an important characteristic. When implementing the hash function, its memory footprint and, when implemented in hardware, resources of the hardware required, e.g. logic elements and registers for an FPGA, have to be considered.

Cryptographic properties of a hash function may be limited to a defined length (5-tuple vs. payload) and/or type of input data (entropy of passwords vs. random data). Furthermore, the function's collision resistance has to be taken into consideration. Finally, different applications may have different constraints regarding the length of the produced output hash. While a short HMAC included in an Ethernet frame causes only minor packet overhead, it can negatively impact its effectiveness.

B. Hash function micro-benchmarks

CPU system We investigate the individual latency of the hash algorithm implementations. Each hash function was executed multiple times for input data lengths ranging from

Hash algorithm	Cycles per B	Fixed cycles per packet	Cycles for 64 B	Output length (bit)
CRC32	0.32	0.00	10.79	32
Checksum	0.44	0.00	30.06	16
SipHash-2-4	1.06	56.40	121.10	64
Poly1305-AES	1.69	83.71	170.38	128
BLAKE2b	3.14	35.85	232.77	8-512
HMAC-SHA512	3.70	1454.51	1578.14	512
HMAC-SHA256	5.57	959.69	1462.13	256

Table I: Hash function latency on CPU DuT

Hash algorithm	Block Size in B	Cycles per Block	Fixed Cycles per Packet	Clock Frequency
SHA3-512	72	12	10	165 MHz
SipHash-2-4	8	3	8	200 MHz

Table II: Hash function latency on FPGA DuT

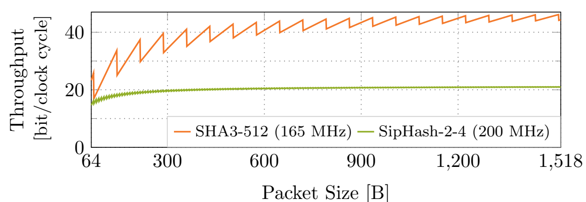


Figure 2: Throughput of hashing FPGA IP cores

2 B to 1500 B on the CPU DuT. For each run, we counted the number of CPU cycles using the timestamp counter (TSC). To model the latency behavior of each hash function, we then performed a linear regression based on the input data lengths and the number of cycles consumed by each algorithm.

The number of fixed CPU cycles per packet reported in Table I highlights that some algorithms are better suited to process small input data such as network packets than others. Especially when comparing HMAC-SHA256 and HMAC-SHA512 to the other cryptographic functions, an increase of per-packet fixed cycles by a factor of up to 40 can be seen. To process 14.88 Mpps for 10GbE with 64 B packets on a single CPU core clocked at 2.40 GHz, the processing for each packet must be completed in 161 CPU cycles. Only the non- or pseudo-cryptographic functions not using an HMAC mode satisfy this requirement. SipHash-2-4 shows the most promising results, being optimized for hashing on 64 bit CPU architectures.

FPGA We evaluate the hash IP cores in our NetFPGA SUME implementation through RTL simulations. Table II and Figure 2 present observed latency and throughput. SHA3-512 hash values are calculated on 72 B data blocks, SipHash-2-4 operates on smaller 8 B blocks. If the input data must be padded to fill the content of the last block, calculation efficiency (i.e. B/clock cycle) decreases and throughput drops. However, this is barely visible for SipHash-2-4 calculation due to the small block size. While the theoretical maximum throughput of the SHA3-512 IP core is 48 bit/clock cycle for infinitely long input data, we observe a maximum rate of 46.14 bit/clock cycle for packets between 64 B and 1518 B. The maximum throughput of the SipHash-2-4 IP core for these packet lengths is 21.02 bit/clock cycle, falling slightly below the theoretical maximum of 21.33 bit/clock cycle. Although the throughput of the SHA3-512 calculation is significantly higher, we were unable to operate the integrated IP core at clock frequencies exceeding 165 MHz. While the SipHash-2-4 logic can be operated at 200 MHz, matching the frequency of the P4->NetFPGA pipeline, we had to place the SHA3-512 IP core in a separate clock domain.

C. Hashing complete packets

For the use case of communication integrity and authentication, we evaluate the hashing of complete packets. For each platform, we perform a baseline measurement, where the P4 program is a simple L2 forwarder without performing any hashing operations.

Throughput Results for maximum throughput are presented in Table III. Independent of packet size, all three platforms reach 10 Gbit/s in the baseline scenario, with the exception for minimum-sized packets on the CPU target. Adding the calculation of hashes reduces the maximum performance such that no platform can reach line rate for packets with minimum size. In our evaluation, the best results are achieved by the Netronome card. Experiments with Checksum and CRC32 as hash algorithms showed that the card can hash packets at line rate regardless of packet size (not shown in Table III). Using SipHash-2-4 about 75 % of line rate for minimum-sized packets can be achieved. Despite high throughput for packet sizes up to 900 B, performance degrades rapidly for larger packets. This behavior can be explained by the SmartNIC's RAM architecture [30]. Our experiments showed that buffers residing in a fast memory region are only used for packets smaller than 900 B for payload processing. For larger packets, slower shared RAM has to be accessed, causing a drop in throughput to approx. 10^{-6}% line rate.

The NetFPGA SUME platform achieves an almost constant

Algorithm	64 B	96 B	128 B	512 B	1024 B	1500 B
t4p4s						
Baseline	95.03	100	100	100	100	100
SipHash-2-4	36.09	46.01	54.73	100	99.17	100
HMAC-SHA512	8.47	11.69	11.11	24.26	31.67	37.80
NFP-4000						
Baseline	100	100	100	100	100	100
SipHash-2-4	75.60	80.71	91.61	99.15	10^{-6}	10^{-6}
NetFPGA SUME						
Baseline	100	100	100	100	100	100
SipHash-2-4	42.00	42.18	42.29	42.56	42.61	42.52
SHA3-512	48.21	42.53	54.27	65.02	71.78	76.00

Table III: Achievable throughput for hashing frames of different sizes in percent, relative to 10 GbE line rate

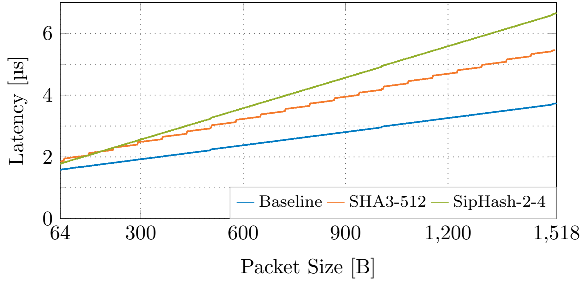


Figure 3: Median latency for NetFPGA (2.5 Gbit/s)

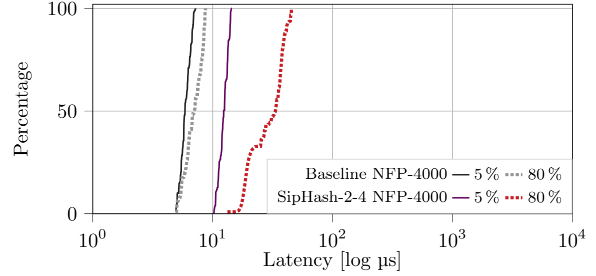


Figure 5: Latency distribution at 5 % and 80 % of respective maximum throughput using 64 B packets

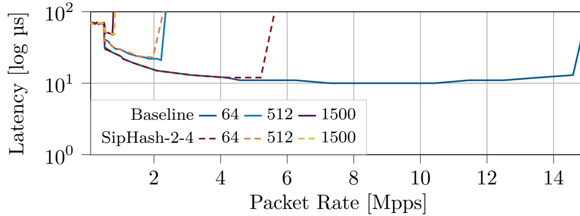


Figure 4: Median latency for CPU system

	LUTs		Registers		BRAM	
	Abs.	%	Abs.	%	Abs. [kB]	%
Baseline	64,533	14.90	109,783	12.67	16,362	30.92
SipHash-2-4	66,380	15.32	114,282	13.19	17,460	32.99
SHA3-512	73,449	16.95	118,689	13.70	17,460	32.99

Table IV: Resource utilization for the NetFPGA SUME

performance of approx. 42 % line rate using SipHash-2-4. The SHA3-512 IP core is clocked slower, but its higher per-cycle throughput results in superior performance for all packet sizes. It reaches 76 % line rate for 1500 B packets. The non-monotonic increase of throughput is caused by the block-based hash calculation. While our prototype is limited to open-source hash implementations, we note that higher throughput could be achieved with commercial IP cores.

The worst performance is shown by the CPU target. Compared to the baseline, for SipHash-2-4 the throughput is more than halved for small packet sizes, roughly matching the calculated latency shown in Table I. Only for packets larger than 390 B line rate is reached. Due to the large number of fixed cycles per packet, SHA512 when used in HMAC mode processes less than 10% line rate for minimum-sized packets and even for large packets is unable to reach line rate.

Latency Figure 3 shows our latency measurements for the NetFPGA SUME. As expected, latency increases linearly with packet size with slight discontinuities due to the block-based hash calculation. We found that for each packet size the measured values do not differ by more than 100 ns.

For t4p4s latency is influenced by the packet rate (see Figure 4, results for HMAC-SHA512 omitted due to low maximum packet rates) as packets are sent either when a burst size of 32 is reached, or after a timeout. This causes increased latency for packet rates below 0.5 Mpps as the batch is not filled quickly enough, instead waiting for the timeout. The latency is independent of the algorithm used, however, increases with packet size (l) due to the increased serialization delay: $t_s(l) = ((l - 64) \cdot \mathrm{b}_{\text{size}})/(t_{s10\text{GbE}})$ in relation to 64 B packets, with $\mathrm{b}_{\text{size}} = 32$ and $t_{s10\text{GbE}} = 1.25ns$. Overall the latency is between 10 µs and 80 µs, however, outliers, which regularly occur when using the DPDK, exist (see Figure 5).

The NPU demonstrates stable behavior below 10 µs with no outliers for the baseline scenario. Performing the SipHash-2-4 operation shifts the latency distribution to the right up to 30 µs and increases the long tail.

Resource Consumption Packet processing in general is parallelizable, scaling well using multi-queue NICs and multi-core CPUs. Thus, the hardware of CPU-based systems can be tailored to meet an application's resource requirements.

Apart from the described performance issues, we did not encounter resource restrictions for the Netronome card as the P4 program is of small size even when adding the SipHash implementation. For other applications, the program may be too large such that the generated firmware image can no longer be loaded onto the card.

Finally, Table IV lists the resource consumption (LUTs, registers, BRAM) of the FPGA-based implementations. Adding hashing functionalities increases resource consumption only moderately by no more than approx. 2 %.

VI. Limitations

The performance of the evaluated platforms depends on the chosen hash function and their implementation. For this work, we selected open-source implementations, because we are primarily interested in the general feasibility of using cryptographic hash functions in programmable data planes. While we have shown that this is possible, more sophisticated hash function implementations (e.g. commercial FPGA IP cores) in combination with an optimized integration into the P4 program (e.g. parallelization, pipelining) could reduce implementation artifacts, improving the performance and resource utilization, but would require further monetary costs and engineering effort.

VII. Conclusion

Our review of the current use of hash functions in P4 applications reveals two insights. First, a prevalent use of CRC, making applications vulnerable to potential attacks targeting hash collisions. Second, protocols and applications requiring cryptographic hashes for authentication or integrity cannot be described using P4. Therefore, the implementation of cryptographic hash functions would increase the applicability of P4 to a wider range of use cases.

We describe prototype implementations integrating cryptographic hashing algorithms in three different P4 target platforms – CPU, NPU, and FPGA. Our analysis shows that the CPU target is easily extensible, but has the highest worse-case latency of up to several milliseconds. The tested NPU offers the highest throughput, but cannot process packets larger than 900 B efficiently. The FPGA-based target offers the lowest latency with small variance. However, the hashing IP core currently cannot be integrated using native P4 features, limiting the programmability and requiring a change of the P4 switch model.

Our measurements show hashing performance to be highly target, algorithm, and use case specific. Therefore, we cannot recommend a one-size-fits-all solution. We rather suggest that P4 targets should implement hash functions – operating on header and payload data – from a family of algorithms, which should be recommended by the P4 specification. These recommendations should include cryptographic hashes and take into account the unique characteristics of platforms such as CPU, NPU, FPGA, or even future ASICs.

Acknowledgment

This work was supported by the German Research Foundation (project ModANet under grant no. CA595/11-1) and by the German-French Academy for the Industry of the Future. The authors would like to thank Fabian Pusch for contributions to our FPGA prototype, Philipp Hagenlocher for the integration and evaluation of various hash functions in t4p4s, as well as the anonymous reviewers for their valuable feedback.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, Jul. 2014.

[2] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016.

[3] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017.

[4] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017.

[5] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks," *IEEE Transactions on Computers*, vol. 62, no. 5, May 2013.

[6] M. Molina, S. Niccolini, and N. Duffield, "A Comparative Experimental Study of Hash Functions Applied to Packet Sampling," in *International Teletraffic Congress (ITC-19)*, 2005.

[7] C. Henke, C. Schmoll, and T. Zseby, "Empirical Evaluation of Hash Functions for Multipoint Measurements," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, Jul. 2008.

[8] R. Jenkins, "A Hash Function for Hash Table Lookup," 1997. [Online]. Available: http://www.burtleburtle.net/bob/hash/doobs.html

[9] N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto Hardware Hash Functions for High Performance Networking ASICs," in *Proceedings of the 2011 ACM/IEEE 7th Symposium on Architectures for Networking and Communications Systems*, Oct. 2011.

[10] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data Plane Performance Diagnosis of TCP," in *Proceedings of the Symposium on SDN Research (SOSR'17)*. ACM Press, 2017.

[11] E. Cidon, S. Choi, S. Katti, and N. McKeown, "AppSwitch: Application-layer Load Balancing Within a Software Switch," in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet'17. New York, NY, USA: ACM, 2017.

[12] J. Kučera, D. A. Popescu, G. Antichi, J. Kořenek, and A. W. Moore, "Seek and Push: Detecting Large Traffic Aggregates in the Dataplane," 2018.

[13] J.-H. Choi, S.-G. Min, and Y.-H. Han, "MACsec Extension over Software-Defined Networks for in-Vehicle Secure Communication," in *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, 2018.

[14] B. Carnevale, L. Fanucci, S. Bisase, and H. Hunjan, "Macsec-based security for automotive ethernet backbones," *Journal of Circuits, Systems and Computers*, vol. 27, no. 05, 2018.

[15] E. Heidinger, C. Heller, A. Klein, and S. Schneele, "Quality of service IP cabin infrastructure," in *29th Digital Avionics Systems Conference*. IEEE, 2010.

[16] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-SDN," *arXiv preprint arXiv:1904.07088*, 2019.

[17] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-IPsec: Implementation of IPsec Gateways in P4 with SDN Control for Host-to-Site Scenarios," *arXiv preprint arXiv:1907.03593*, 2019.

[18] Z. Cao, Z. Wang, and E. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing," in *IEEE INFOCOM 2000*, 2000.

[19] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *USENIX Security Symposium*, 2003.

[20] S. Goldberg and J. Rexford, "Security Vulnerabilities and Solutions for Packet Sampling," in *Proceedings of the 2007 IEEE Sarnoff Symposium*, Apr. 2007.

[21] J.-P. Aumasson and D. J. Bernstein, "SipHash: A Fast Short-Input PRF," in *Progress in Cryptology - INDOCRYPT 2012*, S. Galbraith and M. Nandi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[22] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: Simpler, Smaller, Fast as MD5," in *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2013.

[23] P. Hoffman and B. Schneier, "Attacks on Cryptographic Hashes in Internet Protocols," RFC Editor, RFC 4270, Nov. 2005.

[24] W. M. Eddy, "TCP SYN flooding attacks and common mitigations," RFC 4987, 2007.

[25] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel, "High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016.

[26] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[27] Austin Appleby, "Poly1305 git repository," 2016. [Online]. Available: https://github.com/floodyberry/poly1305-donna

[28] "NFP-4000 Theory of Operation," Netronome Systems Inc., Tech. Rep., 01 2016, last accessed: 2019-06-17. [Online]. Available: https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf

[29] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[30] S. Wray, "The Joy of Micro-C," 2014, https://open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf.

A.3.2 Rapid Prototyping of Packet Processing Devices for Aeronautical Applications

This work was published in *Proceedings of the 6th International Workshop on Aircraft System Technologies*, 2017 [73].

RAPID PROTOTYPING OF PACKET PROCESSING DEVICES FOR AERONAUTICAL APPLICATIONS

Fabien Geyer[1], Max Winkel[1], Stefan Schneele[1]

[1]Airbus Group Innovations, TX2P
Willy-Messerschmitt-Str. 1, 81663 Munich, Germany

fabien.geyer@airbus.com +49 89 607 23625

Abstract
Recent developments made in the area of network-specific reconfigurable hardware and associated description language – namely the P4 language and its back-ends – promise interesting features for rapid prototyping of packet processing devices. Due to the need for high flexibility and increasing trend towards softwarization, such solution is of interest for the aeronautical industry. Our contributions in this paper are two fold. An analysis of the functionalities of P4 with respect to requirements usually necessary for applications and network protocols in the aeronautic industry is first performed. In a second step a performance evaluation of an existing software-based back-end using Intel DPDK is performed and compared to existing hardware solutions.

1 INTRODUCTION

In the last two decades, distributed embedded electronic applications have become the norm in a large part of the aeronautical industry. Those applications cover a large set of functionalities with different requirements, ranging from flight control with hard real-time and strict safety constraints, to passenger entertainment with less stringent constraints. Due to those constraints and safety aspects associated with aircrafts, specific equipments are generally used in order to fulfill those constraints. When such equipments are not available off-the-shelves, costly and time consuming developments have to be undertaken. This is especially true in the scope of networking equipments, since standard off-the-shelves devices for networking usually do not support aeronautical-specific network protocols or safety-related functionalities.

A prevailing solution used to address this issue is to employ FPGAs (Field Programmable Gate Arrays) since they offer high customizability with high performance.

The two main drawbacks of this approach are that current developments using FPGAs require a high level of expertise and long development times to produce efficient and bug-free devices. We propose in this paper to look at recent developments made in the area of network-specific reconfigurable hardware and associated tools, namely P4, recently proposed by Bosshart et al. in [1], and assess if they are a fit for aeronautical applications.

Our main contribution in this paper is an analysis of those new solutions in the scope of aeronautical applications in term of offered features and performance. We first investigate their applicability from a functional point of view and identify missing features of the current approaches. Then on a more practical point of view, we do a performance analysis using measurements on a target hardware and investigate if those new developments are sufficient for aeronautical applications from a performance point of view.

This rest of this paper is organized as follows. In Section 2 we present similar research studies. We then introduce in Section 3 the new advances made regarding network-specific reconfigurable hardware and their associated tools. In Section 4, we present its applicability to aeronautical requirements, with a concrete application to existing aeronautical network protocol and architectures. We do a performance evaluation of a target hardware in Section 5 with results regarding packet processing latency of frames and resource utilization. Finally, Section 6 concludes our work.

2 RELATED WORK

Approaches towards a top-down description of data-plane in a high-level programming language have been proposed since the late 1990s and early 2000nd. Kohler et al. proposed Click in [2] which enables flexible packet processing in software, but with the drawback of difficulty regarding compilation to dedicated hardware.

More recently with the increasing use of FPGAs (Field Programmable Gate Array) for packet processing, Brebner and Jiang proposed the PX programming language in [3] with a compiler targeting FPGAs. Dedicated hardware for packet processing such as NPUs (Network Processor Unit) [4] or RMT (Reconfigurable Match Table) [5] have also been proposed. Song proposed POF (Protocol-Oblivious Forwarding) in [6], which defines an Flow Instruction Set which is used for processing packets.

Regarding purely software-based packet processing on commodity multi-core processors, various works have been performed on the performance of such platforms. Dobrescu et al. evaluated the predictability of such platform in [7]. They evaluated how contention for shared hardware resources such as caches can be taken into account for improving performance predictability, an important aspect in case of safety critical applications. More recently, Emmerich et al. benchmarked various Linux-based software stacks for software-based packet processing in [8] and identified various bottlenecks responsible for poor performance.

3 A NEW APPROACH FOR PACKET PROCESSING DEVICES

3.1 Main promises of P4

In conjunction with the current trend towards softwarization of functionalities in the field of communication networks with the advent of Software Defined Networking and related technologies, a recent development called P4 [1] – Programming Protocol-Independent Packet Processor – proposes a flexible way to specify packet processing devices. The main promises of the P4 programming language and toolchain are:

1. A simple specification of packet processing pipelines using a high-level Domain Specific Language (DSL), requiring no expert knowledge about the final hardware. This DSL was specially designed to be expressive enough for the various actions necessary in network protocols, while restrictive enough to be enable simple compilation to dedicated target hardware. Examples of P4 descriptions are given in Listings 1 and 2. The complete specification of the P4 language is available on the P4 website [9].
2. Compilation of specification for different hardware targets, ranging from FPGAs (Field Programmable Gate Array) to NPUs (Network Processing Unit) to finally purely software solutions targeting multi-core and many-core processors;
3. Reconfigurability in order to modify the behavior of packet-processing devices in the field;
4. Possibility to test packet processing pipelines using well-known network emulation tools such as mininet [10] and ability to emulate complete network architectures.

This approach is also in line with model driven engineering, where high level descriptions of systems are used in order to formally verify various properties of systems.

Listing 1 – Example of Ethernet frame format definition in P4

```
header_type ethernet_t {
  fields {
    dstAddr   : 48;
    srcAddr   : 48;
    etherType : 16;
  }
}
```

Listing 2 – Example of IPv4 packet routing in P4

```
action route_ipv4(dst_port, dst_mac, src_mac, vid) {
  modify_field(standard_metadata.egress_spec, dst_port);
  modify_field(ethernet.dst_addr, dst_mac);
  modify_field(ethernet.src_addr, src_mac);
  modify_field(vlan_tag.vid, vid);
  add_to_field(ipv4.ttl, -1);
}
```

3.2 Abstract forwarding model

P4 uses a generic packet processing pipeline as a basis called *abstract forwarding model*. This model applied to a switch is illustrated here in Figure 1. Packets are first parsed according to customizable frame format definitions.

Based on the fields and associated values of the protocols, so-called *match+action tables* are used in order to process packets. Available actions include packet modification (changing filed value, adding or removing headers), replication (for broadcast or multicast), drop packets, triggering of flow control (namely update of action tables such as counters or policers). Those match+action tables are conceptually similar to the ones used in OpenFlow switches.

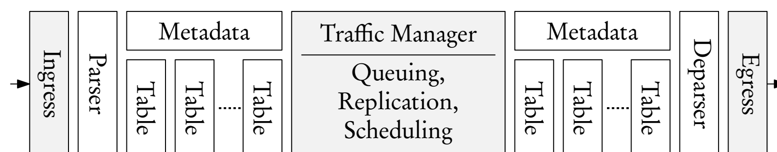


Figure 1 – P4 Abstract Forwarding Model of a switch

4 APPLICABILITY FOR AERONAUTICAL APPLICATIONS

As illustrated in Section 3, P4 promises various properties which make it attractive for the aeronautical industry. We investigate in this section the current features of P4 and their applicability to aeronautical applications.

4.1 The good parts of P4

The main advantage of P4 is the decorrelation between the behavior of a packet processing device and the hardware which is used. It means that engineers are not tied to a specific set of network protocols implemented by hardware vendors. This is especially relevant in the aeronautical industry since the two following constraints are usually present: 1) Specific network protocols only used by the aeronautical industry are used (e.g. ARINC standards); 2) For safety reasons, devices must usually only implement the required protocols and functionalities, meaning that no additional features should be implemented or used. Those two constraints usually prevents COTS (Commercial Off-The-Shelf) devices to be used since they may not support the required protocols, or implement a larger set of protocols than the ones which are required. With P4, the usability of COTS devices increases.

The second advantage of P4 is the simplicity and constraints put on the abstract forwarding model presented earlier in Figure 1. Since P4 forbids dynamic memory allocation and iterations with unknown counts – unlike more generic programming languages such as C – formal derivations of worst-case execution time and resource usage of a P4 program are fairly straightforward. This means that per-packet latency, memory footprint and maximum throughput of a packet-processing pipeline can be

determined at compile time. This is again relevant in the aeronautical industry since constraints on those cost factors are required in real-time applications.

Finally, regarding the features supported by P4 in term of packet processing actions, it covers most of the use-cases relevant for network protocols used by the aeronautical industry. Missing features are listed in the next section.

4.2 Avenues for improvement

While P4 offers a lot of flexibility for expressing packet-processing pipelines, some features are still missing for more advanced uses needed in avionic applications.

Egress packet scheduling cannot be directly described by P4. While there is some limited support for defining the priority of a packet in case of targets supporting Strict Priority Queuing (SPQ), more advanced schedulers such as Weighted Fair Queuing [11] or Deficit Round Robing [12] cannot be defined or configured via P4. In other words, the description of more advanced Quality-of-Service architectures which are envisioned for next-generation aeronautical backbones such as the one presented in [13] is limited with P4.

Since safety is an important aspect of aeronautical applications, specification and programing languages need to have defined behavior. In the current specification of P4 [9], some aspects are incompletely specified, as for instance overflow of integers, casting between different data types, exception handling, and initial values of table entries and packet attributes.

Finally, time-based or time-triggered protocols cannot be directly described using P4 since there are no primitives for describing access to a clocking information. This drawback prevents the implementation of time-synchronization protocols for packet timestamping, or egress scheduling based on time information.

We note that the drawbacks listed here are with respect to the current specification of P4 [9] and P4 is under active development and research. For instance, a solution for the specification of egress packet scheduling has been recently proposed by Sivaraman et al. in [14]. Similarly, some issues of undefined behaviors have been addressed in the new version of the P4 specification.

5 PERFORMANCE EVALUATION

5.1 Presentation

We propose in this section to do a performance evaluation of a P4 target proposed by Laki et al. in [15]. This purely software target was chosen since access to dedicated P4 hardware was not available or possible at the moment of writing. This target is based on the Intel Data Plane Development Kit framework (DPDK) [16], which is a set of libraries and drivers for fast packet processing in the Linux userland.

Our measurement setup is presented in Figure 2. Traffic up to 4 Gbit/s is generated by an Anritsu Network Analyzer MD1230B [17] on four different 1 Gbit/s Ethernet links. This traffic is then processed and forwarded to the according output port on

a standard PC equipped with an Intel i7-2600 CPU[1] with 4 physical cores and an Intel I340 T4 network card[2]. Regarding software, Ubuntu 16.04 with the default kernel, DPDK 16.04 and the software presented in [15][3] was used.

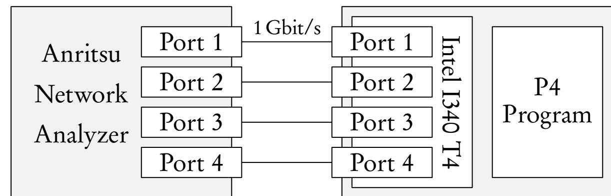


Figure 2 – Measurement setup used for the performance evaluation

Regarding the P4 program which was used for making the measurements, we used here the simple MAC-learning Ethernet switch available as an example with the code from [15]. This P4 program parses the Ethernet header and decides which output port should be used based on a learned MAC table. Minor modifications to the code from [15] were made to add profiling and remove some unnecessary overhead.

5.2 Framerate

Figure 3 presents the framerate achieved by the platform for different packet sizes. While the platform is able to support most of the load, we notice that as we reach 95 % to 100 % line load, packet drops start to occur. This means that the P4 switch is almost able to process the 4 Gbit/s of data sent.

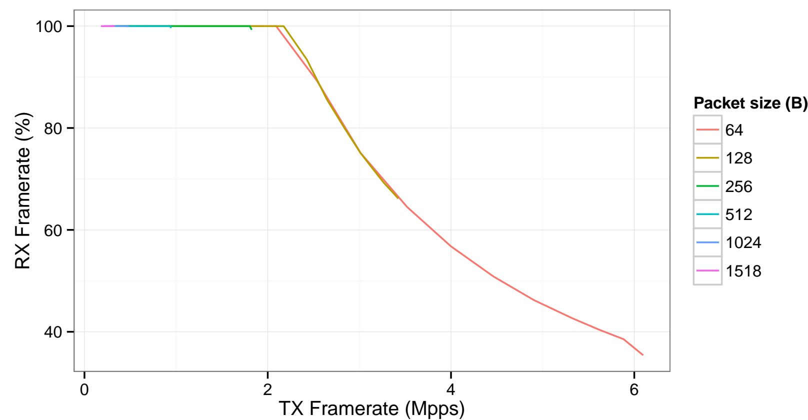


Figure 3 – Framerate processed by the P4 switch

[1]Intel i7-2600: `https://ark.intel.com/products/52213`
[2]Intel I340 T4: `https://ark.intel.com/products/49186`
[3]P4@ELTE software from [15]: `https://github.com/P4ELTE/p4c`

5.3 Packet processing latency

Figure 4 presents the packet processing latency as a function of the time between two frames (or framegap). We notice that for framegaps larger than 1 µs the processing latency is of 24 µs for packet sizes of 1518 B. For framegaps smaller than 1 µs the processing latency increases up to 1 ms depending on the packet size. This means that some bottleneck is hit, as already shown before in Figure 3.

A comparison between this value of 24 µs and previous work [18] done on an industrial AFDX switch and an HP E3800 switch is presented in Table 1.

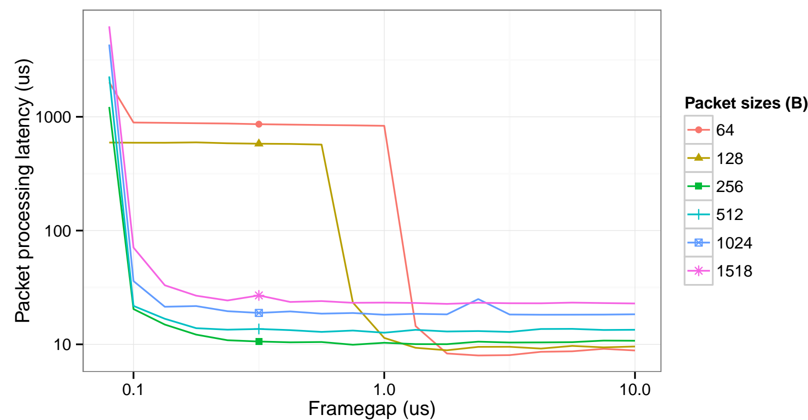


Figure 4 – Packet processing latency as a function of the time between two frames

Switch	Proc. latency
Rockwell Collins AFDX switch	5 µs
HP E3800 with hardware switching and without OpenFlow	7.2 µs
HP E3800 with hardware switching and OpenFlow	7.7 µs
HP E3800 with software switching	613 µs *(average)*
P4 software switch with DPDK from Section 5	24 µs

Table 1 – Comparison of packet processing latency of the evaluated P4 switch with numerical results from [18]

5.4 Profiling

Figure 5 presents the profiling of the P4 program using the `operf`[4] statistical profiler for Linux, namely how much time is spent in each function of the P4 program and the system. Note that only 2 cores of the CPU are used, explaining the maximum value of 200 %. Three different function groups are presented in Figure 5:

[4] `http://oprofile.sourceforge.net`

- P4 primitives: *Parser*, *Table* and *Actions*, which is the part taking the most resources (up to 70 %);
- DPDK primitives: *Ethernet Driver*, *Ethernet Library* and *Run Time Environment (RTE)*, which take relatively low resources compared to the P4 primitives;
- Other functions: the C standard library (`libc`), Linux kernel, and overhead.

While in the tested setup the P4 program is able to almost fully process the 4 Gbit/s of traffic as shown in Section 5.2, some work on reducing the P4 resources must be done in case additional ports would be used.

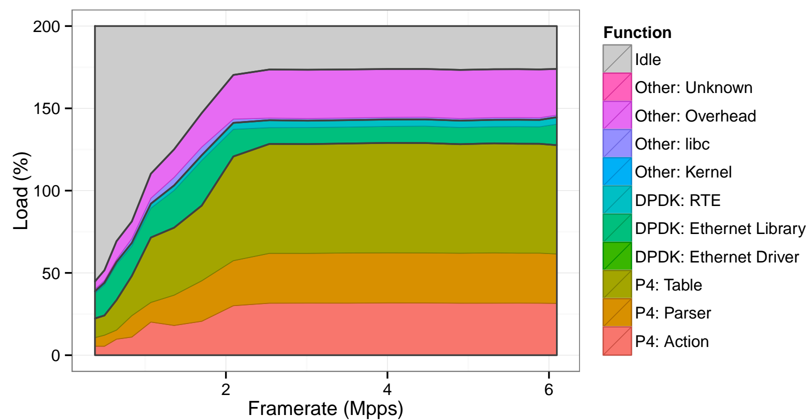


Figure 5 – Profiling of the Linux computer with 64 B *packet sizes*

6 CONCLUSION

We investigated in this paper recent developments made in the area of network-specific reconfigurable hardware and associated description language, namely the P4 programing language. This new development allows faster development of customized packet processing devices such as Ethernet switches or routers without the need for a deep knowledge about the target hardware architecture.

We presented in this paper P4 and its functionalities. We showed that its high flexibility in combination with simple building block enabling a formal analysis make it an attractive platform for some network protocols used in aeronautical use-cases. However, some features such as definition of egress packet scheduling and method for time-based or time-triggered protocols are still lacking for more advanced network protocols. Finally, a performance evaluation was carried out on a purely software-based target with a comparison with an aeronautical and a COTS switch.

Since P4 is still under active development and lacks some important features mentioned previously, it is not yet ready for production use in aeronautical use-cases.

Nevertheless, such platform is of high importance in the area of prototyping where functional validation and some indication about performance evaluation are necessary. Its simple cost model and associated formal analysis also make it a good target for future certification of packet processing devices.

7 REFERENCES

References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[2] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[3] Gordon Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro*, 34(1):8–18, January 2014.

[4] Ran Giladi. *Network Processors: Architecture, Programming, and Implementation*. Morgan Kaufmann, 2008.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110, 2013.

[6] Haoyu Song. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in Software Defined Networking*, pages 127–132, 2013.

[7] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 141–154, April 2012.

[8] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems. *Proceedings of the 14th International Conference on Networks (ICN 2015)*, pages 78–83, April 2015.

[9] The P4 Language Consortium. The P4 Language Specification. Version 1.0.3, November 2016. URL `http://p4.org/spec/`.

[10] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6. ACM, October 2010.

[11] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, August 1989.

[12] M. Shreedhar and George Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[13] Fabien Geyer, Stefan Schneele, Matthias Heinisch, and Peter Klose. Simulation and Performance Evaluation of an Aircraft Cabin Network Node. In Frank Thielecke Otto von Estorff, editor, *Proceedings of the 4th International Workshop on Aircraft System Technologies*, AST 2013, pages 323–332. Shaker-Verlag, April 2013. ISBN 978-3-8440-1850-9.

[14] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 44–57, 2016.

[15] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 629–630, 2016.

[16] Intel. Intel DPDK: Data Plane Development Kit, Accessed 2016/11/24. URL `http://dpdk.org`.

[17] Anritsu. Data Quality Analyzer MD1230B, Accessed 2016/11/24. URL `http://www.anritsu.com/en-US/test-measurement/products/MD1230B`.

[18] Peter Heise, Fabien Geyer, and Roman Obermaisser. Deterministic OpenFlow: Performance Evaluation of SDN Hardware for Avionic Networks. In *Proceedings of the 11th International Conference on Network and Service Management (CNSM)*, pages 372–377, November 2015.

A.3.3 Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications

This work was published in *Proceedings of the 9th European Congress Embedded Real Time Software and Systems*, 2018 [71].

Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications

Fabien Geyer*
Technical University of Munich (TUM)
D-85748 Garching b. München, Germany
Email: fgeyer@net.in.tum.de

Max Winkel
Airbus Group Innovations
D-81663 Munich, Germany
Email: max-oliver.winkel@airbus.com

***Abstract*—Recent developments made in the area of network-specific reconfigurable hardware and associated description language – namely the P4 language and its back-ends – promise interesting features for rapid prototyping of packet processing devices. Due to the need for high flexibility and increasing trend towards softwarization, such solution is of interest for the aeronautical industry where custom network protocols are generally used.**

Our contributions in this paper are two fold. First, an analysis of the functionalities of P4 with respect to requirements usually necessary for applications and network protocols in the aeronautic industry is performed. In a second step, a performance evaluation of three different platforms was done. Those platforms represent three use-cases: an existing software-based back-end using Intel DPDK, a hardware network accelerator based on a network processor unit, and an FPGA-based platform. Performance of those platforms are compared to existing state-of-the-art hardware solutions used in by the aeronautical industry.

I. Introduction

In the last two decades, distributed embedded electronic applications have become the norm in a large part of the aeronautical industry. Those applications cover a large set of functionalities with different requirements, ranging from flight control with hard real-time and strict safety constraints, to passenger entertainment with less stringent constraints. Due to those constraints and safety aspects associated with aircrafts, specific equipments are generally used in order to fulfill those constraints. When such equipments are not available off-the-shelves, costly and time consuming developments have to be undertaken. This is especially true in the scope of networking equipments, since standard off-the-shelves devices for networking usually do not support aeronautical-specific network protocols or safety-related functionalities.

A prevailing solution commonly used to address this issue is to employ FPGAs (Field Programmable Gate Arrays) since they offer high customizability with high performance at moderate costs. The two main drawbacks of this approach are that current developments using FPGAs require a high level of expertise and long development times to produce efficient and bug-free devices. We propose in this paper to look at recent developments made in the area of network-specific reconfigurable hardware and associated tools, namely P4 (Programming Protocol-independent Packet Processors), recently proposed by Bosshart et al. in [1], and assess if they are a fit for aeronautical applications.

Our main contribution in this paper is an analysis of those new solutions in the scope of aeronautical applications in terms of offered features and performance. We first investigate their applicability from a functional point of view and identify missing features of the current approaches. We show that its high flexibility in combination with simple building blocks enabling a formal analysis make it an attractive platform for some network protocols used in aeronautical use-cases. However, some features such as controlling of egress packet scheduling and methods for time-based or time-triggered protocols for more advanced network protocols are undefined or vendor-specific. In order to evaluate P4 in an aeronautical context, we choose AFDX as a case-study and demonstrate that a simplified AFDX switch can be implemented using P4.

In a second step, we do a performance analysis using measurements on three different target hardware and investigate if those new developments and platform are sufficient for aeronautical applications from a performance point of view. Our first target is a purely software-based solution using Intel's Data Plane Development Kit framework (DPDK) [2], which is a set of libraries and drivers for fast packet processing in the Linux userland. Our second target is based around the Netronome Agilio CX platform, a hardware-based Network Flow Processor (NFP) which is able to offload most packet processing functionalities from the CPU. Our third target is based on a FPGA (Field-Programmable Gate Array), where packet processing is fully performed in hardware. Measurements done using a network analyzer show that those platforms are able to achieve packet processing latencies similar to those of devices used by the aeronautical industry and a commercial-of-the-shelf (COTS) Ethernet switch.

The rest of this paper is organized as follows. In Section II we present similar research studies. We then introduce in Section III the new advances made regarding network-specific reconfigurable hardware and their associated tools. In Section IV, we present its applicability to aeronautical requirements, with a concrete application to existing aeronautical network protocol and architectures. We do a performance evaluation of a target hardware in Section V with results regarding packet processing latency of frames and resource

*This work was performed while the author was with Airbus Group Innovations in Munich, Germany.

utilization. Finally, Section VI concludes our work.

II. Related work

Approaches towards a top-down description of data-plane in a high-level programming language have been proposed since the late 1990s and early 2000nd. Kohler et al. proposed Click in [3] which enables flexible packet processing in software, but with the drawback of difficulty regarding compilation to dedicated hardware.

More recently with the increasing use of FPGAs (Field Programmable Gate Array) for packet processing, Brebner and Jiang proposed the PX programming language in [4] with a compiler targeting FPGAs. Dedicated hardware for packet processing such as NPUs (Network Processor Unit) [5] or RMT (Reconfigurable Match Table) [6] have also been proposed. Song proposed POF (Protocol-Oblivious Forwarding) in [7], which defines an Flow Instruction Set which is used for processing packets.

Regarding purely software-based packet processing on commodity multi-core processors, various works have been performed on the performance of such platforms. Dobrescu et al. evaluated the predictability of such platform in [8]. They evaluated how contention for shared hardware resources such as caches can be taken into account for improving performance predictability, an important aspect in case of safety critical applications. More recently, Emmerich et al. benchmarked various Linux-based software stacks for software-based packet processing in [9] and identified various bottlenecks responsible for poor performance.

On the proposition of more advanced networking stacks for industrial applications, various work have been done in the scope of Quality-of-Service and auto-configuration. Henneke et al. provided a survey over the challenges and proposed solutions on applying Software-Defined Networking (SDN) paradigms to industrial networks in [10]. Various requirements such as application-aware QoS, timing performance, monitoring, security, reliability were reviewed, with the conclusion that experience on applying SDN to existing industrial networks is still lacking. Heise et al. proposed to apply SDN paradigms to avionic networks with real-time guarantees in [11]. It was showed that deterministic network functionalities similar to the one commonly found in real-time networks could be achieved using SDN and OpenFlow. While those works have shown the possible applicability of SDN to industrial networks, a P4-based solution as presented in this paper might be more tailored to embedded applications where simpler functionalities are required.

III. A new approach for packet processing devices

A. Main promises of P4

In conjunction with the current trend towards softwarization of functionalities in the field of communication networks with the advent of Software Defined Networking and related technologies, a recent development called P4 [1] – Programming Protocol-Independent Packet Processor – proposes a flexible way to specify packet processing devices. The main promises of the P4 programming language and toolchain are:

1) A simple specification of packet processing pipelines using a high-level Domain Specific Language (DSL), requiring no expert knowledge about the final hardware. This DSL was specially designed to be expressive enough for the various actions necessary in network protocols, while restrictive enough to enable simple compilation to dedicated target hardwares. Sample snippets of P4 descriptions for standard Ethernet and IPv4 routing are given in Listings 1 and 2. The complete specification of the P4 language is available on the P4 website [12, 13].
2) Compilation of specification for different hardware targets, ranging from FPGAs (Field Programmable Gate Array) to NPUs (Network Processing Unit) to finally purely software solutions targeting multi-core and many-core processors, as presented later in Section V;
3) Reconfigurability in order to modify the behavior of packet-processing devices in the field;
4) Possibility to test packet processing pipelines using well-known network emulation tools such as mininet [14] and ability to emulate complete network architectures.

This approach is also in line with model driven engineering, where high level descriptions of systems are used in order to formally verify various properties of systems.

Currently, two different P4 standards are evolving in parallel: $P4_{14}$, which is the original P4 and subject of this paper, and $P4_{16}$, a major redesign of the language with an object oriented approach. If not stated otherwise, the text refers to $P4_{14}$ only.

Listing 1: Example of Ethernet frame format definition in P4

```
header_type ethernet_t {
  fields {
    dstAddr   : 48;
    srcAddr   : 48;
    etherType : 16;
  }
}
```

Listing 2: Example of IPv4 packet routing in P4

```
action route_ipv4(dst_port, dst_mac, src_mac, vid) {
  modify_field(standard_metadata.egress_spec, dst_port);
  modify_field(ethernet.dst_addr, dst_mac);
  modify_field(ethernet.src_addr, src_mac);
  modify_field(vlan_tag.vid, vid);
  add_to_field(ipv4.ttl, -1);
}
```

B. Abstract forwarding model

P4 uses a generic packet processing pipeline as a basis called *abstract forwarding model*. This model applied to a switch is illustrated here in Figure 1. Packets are first parsed according to customizable frame format definitions.

Based on the fields and associated values of the protocols, so-called *match+action tables* are used in order to process packets. Available actions include packet modification (changing field value, adding or removing headers), replication (for

broadcast or multicast), dropping packets or triggering of flow control (namely update of action tables such as counters or policers). Those match+action tables are conceptually similar to the ones used in OpenFlow switches.

IV. Applicability for aeronautical applications

As illustrated in Section III, P4 promises various properties which make it attractive for the aeronautical industry. We investigate in this section the current features of P4 and their applicability to aeronautical applications.

A. The good parts of P4

The main advantage of P4 is the decorrelation between the behavior of a packet processing device and the hardware which is used. It means that engineers are not tied to a specific set of network protocols implemented by hardware vendors. This is especially relevant in the aeronautical industry since the two following constraints are usually present:

1) Specific network protocols only used by the aeronautical industry are used (e.g. ARINC standards);
2) For safety reasons, devices must usually only implement the required protocols and functionalities, meaning that no additional features should be implemented or used.

Those two constraints usually prevent COTS devices to be used since they may not support the required protocols, or implement a larger set of protocols than the ones which are required. With P4, the usability of COTS devices increases.

The second advantage of P4 is the simplicity and constraints put on the abstract forwarding model presented earlier in Figure 1. Since P4 forbids dynamic memory allocation and iterations with unknown counts – unlike more generic programming languages such as C – formal derivations of worst-case execution time and resource usage of a P4 program are fairly straightforward. This means that per-packet latency, memory footprint and maximum throughput of a packet-processing pipeline can be determined at compile time. This is again relevant in the aeronautical industry since constraints on those cost factors are required in real-time applications. In conjunction with FPGA based platforms for P4 such as [15], deterministic processing may be achieved in hardware components.

Finally, regarding the features supported by P4 in terms of packet processing actions, it covers most of the use-cases relevant for network protocols used by the aeronautical industry. Missing features are listed in the next section.

B. Avenues for improvement

While P4 offers a lot of flexibility for expressing packet-processing pipelines, some features are still missing for more advanced uses needed in avionic applications.

Egress packet scheduling cannot be directly described by P4. While there is some limited support for defining the priority of a packet in case of targets supporting Strict Priority Queuing (SPQ), more advanced schedulers such as Weighted Fair Queuing [16] or Deficit Round Robing [17] are not defined in P4. There are vendor-specific interfaces to control the scheduling, such as described in Section V-A3. Nevertheless, in general, the description of more advanced Quality-of-Service architectures which are envisioned for next-generation aeronautical backbones such as the one presented in [18] is limited with P4.

Since safety is an important aspect of aeronautical applications, specification and programing languages need to have defined behavior. In the 2014 specification of P4 [12], some aspects are incompletely specified, as for instance overflow of integers, casting between different data types, exception handling, and initial values of table entries and packet attributes.

Finally, time-based or time-triggered protocols cannot be directly described using P4 since there are no primitives for describing access to a clocking information. This drawback prevents the implementation of time-synchronization protocols for packet timestamping, or egress scheduling based on time information. Such protocols and mechanisms need to implemented around P4 in a target specific manner and may eventually be interfaced, for example, by vendor-specific metadata.

We note that the drawbacks listed here are with respect to the 2014 specification of P4 [12], being the version which is supported by the majority of platforms available at the time of writing. A new version of the language has been published under the name $P4_{16}$ [13], along with the Portable Switch Architecture (PSA) [19] defining a set of standardized common capabilities of network switches. Issues regarding undefined behaviors have been addressed in $P4_{16}$. For functionalities needing timing information, timestamps at ingress and egress have been added in the Portable Switch Architecture with a recommendation to use microsecond precision. The use-cases targeted for this timing information are inband telemetry for measuring queuing latencies, and checking of timeouts or keep-alive in network protocols.

Due to the promises of P4 and its applicability to a large variety of use-cases, improvements have been proposed in the literature. For instance, a solution for the specification of egress packet scheduling has been recently proposed by Sivaraman et al. in [20]. Extensions of P4 switches with other languages are also being investigated in order to simplify the addition of functionalities to switches. For example, the Domino programming language has recently been proposed by Sivaraman et al. in [21].

C. Case-study: AFDX switching

In order to evaluate the applicability of P4 to aeronautical use-cases, we propose to apply P4 to the case-study of Avionics Full-Duplex Switched Ethernet (AFDX), an Ethernet-based protocol for safety-critical applications standardized in ARINC 664 Part 7 [22].

An AFDX network is composed of *end-systems* and *switches* as nodes. End-systems serve as source and destination nodes in the network, over which applications may send data according to bandwidth restrictions to avoid overloading. One fundamental building block of AFDX is the notion of *virtual link* (VL), which can be seen as rate-constrained network

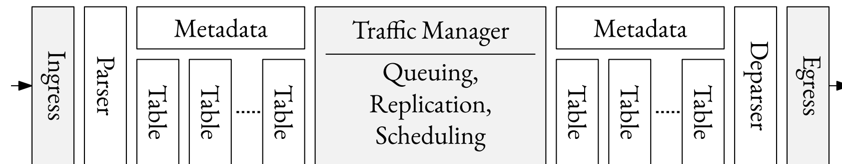


Figure 1: P4 Abstract Forwarding Model of a switch

tunnels. The parameters describing a VL are: the emitter end-system of this VL, the list of receiving end-systems, static routes between emitter and receivers, the Bandwidth Allocation Gap (BAG), as well as minimum and maximum frame length (s_{min} and s_{max}). The BAG is defined as the minimum time interval between the first bit of two consecutive frames from the same VL.

We will focus in the rest of this section on the implementation of a simplified AFDX switch with P4. Switches must ensure the following functionalities for each frame entering the switch:

- Identification of the Virtual Link;
- Frame filtering and policing based on Virtual Link parameters: allowed input port, BAG and frame length limits;
- Forwarding of the frame to the correct output ports based on the Virtual Link routing configuration.

Frames in AFDX are based on the standard Ethernet frame format. An addressing schema is defined in ARINC 664 Part 7 (Section 3.2.5) in order to encode the Virtual Link identification number in the last 16 bits of the destination MAC address. Listing 3 represents how this identifier can be easily extracted using P4.

Listing 3: Simplified AFDX frame header in P4

```
header_type afdx_t {
  fields {
    dstConst   : 32;
    dstVlinkID : 16;
    srcAddr    : 48;
    etherType  : 16;
  }
}
header afdx_t afdx;
```

Switch configuration and routes in P4 are saved in so-called *tables*. For our case-study, we define a table containing the allowed input port of each Virtual Link and its output port, as illustrated in Listing 4. Incoming packets with invalid Virtual Link identifiers are dropped here.

Listing 4: AFDX forwarding table

```
table tbl_forward_virtual_link {
  reads {
    standard_metadata.ingress_port : exact;
    afdx.dst_vlink_id              : exact;
  }
  actions {
    drop;
    forward;
  }
  size : MAX_VIRTUAL_LINKS;
}
```

The P4 function `ingress` is called for each incoming frame. Listing 5 describes a simplified ingress function for AFDX with basic frame integrity checking (with respect to the ARINC 664 requirements) application of the table from Listing 4, and policing.

Listing 5: AFDX ingress function

```
control ingress {
  integrity_check();
  apply(tbl_forward_virtual_link);
  traffic_policing();
}
```

Regarding policing of AFDX frames, a simple process based on validating the BAG timing properties and minimum and maximum frame sizes is described in the ARINC 664 Part 7 standard. While frame sizes validation is possible with P4, filtering based on the time between frames is not (as mentioned earlier in Section IV-B). Since from a functional point of view the goal is to limit the bandwidth of each Virtual Link, the standard policing mechanisms provided by P4 may be used as an alternative. Those mechanisms are based on the use of the two token buckets, as defined in RFC 2698 [23]. Those meters can be easily created in P4, as illustrated in Listing 6.

Listing 6: Policing based on

```
meter vlink_bandwidth_bytes {
  type   : bytes;
  direct : tbl_forward_virtual_link;
  result : scheduling_metadata.color_bytes;
}
```

Finally, the last step is to forward frames to the correct output ports. This is illustrated in Listing 7 with the use of multicast by a vendor-specific mechanism.

Listing 7: forwarding

```
action forward(egress_ports) {
  modify_field(standard_metadata.egress_spec,
        EGRESS_SPEC_MULTICAST);
  modify_field(intrinsic_metadata.egress_port_bitmap,
        egress_ports);
}
```

We listed here only a subset of the functionalities needed by an AFDX switch. More advanced features such as of operational modes, priority-based packet scheduling, and monitoring functionalities based on SNMP can also be implemented using P4.

V. Performance evaluation

A. Presentation

We propose in this section to do a performance evaluation of three different P4 targets: software-based, software-based with hardware acceleration and FPGA-based platform.

1) Software based target: The first target is a purely software based solution, which has been proposed by Laki et al. in [24]. This target is based on the Intel Data Plane Development Kit framework (DPDK) [2], which is a set of libraries and drivers for fast packet processing in the Linux userland. As illustrated in Figure 2, DPDK enables developers to bypass the kernel and process frames directly in user-space. Standard kernel overheads are avoided using DPDK, namely system calls, context switching on blocking I/O, data copying from kernel to user space or interrupt handling in kernel. Benchmarks have shown that DPDK enables much faster packet processing, as shown for example in [9]. Predictable performance without the interference of the Linux scheduler and other processes may be achieved by pinning the DPDK process to dedicated CPU cores.

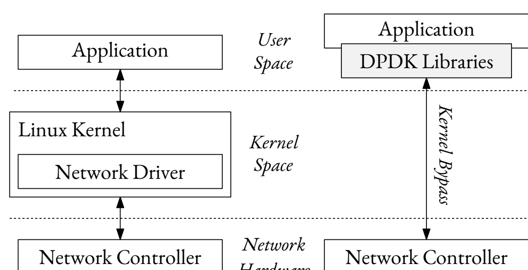


Figure 2: Overview of the DPDK framework

A standard PC equipped with an Intel i7-2600 CPU[2] with 4 physical cores and an Intel I340 T4 network card[3] was used here. This network card has 4 ports supporting 1 Gbit/s Ethernet. Regarding software, Ubuntu 16.04 with the default kernel, DPDK 16.04 and the software presented in [24][4] was used. This software is compiler from P4 to a DPDK-based application. Minor modifications to the code from [24] were made to add profiling and remove some unnecessary overhead.

Such a platform might be interesting for functional testing or in services with short life-cycles and no hard real-time guarantees needed. Typical services such as passenger connectivity could fit these requirements, since on-board passenger devices have a fast update rate, with changing needs and protocols.

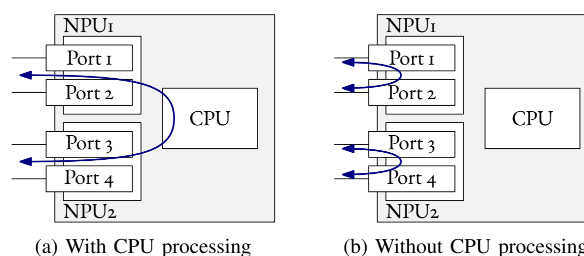
Still, it should be kept in mind, that the performance analysis of the software target was performed on a PC and the results are not directly transferable to embedded platforms with usually very limited resources (CPU processing power, memory, caches, interface bandwidths). If to be used in field, an in-depth performance analysis has to be performed using the specific hardware and software.

[2]Intel i7-2600: https://ark.intel.com/products/52213
[3]Intel I340 T4: https://ark.intel.com/products/49186
[4]P4@ELTE software from [24]: https://github.com/P4ELTE/p4c

2) Network processor platform: The second target is based around the Netronome Agilio CX SmartNIC [25], a hardware-based Network Flow Processor (NFP) which is able to offload the packet processing from the CPU. Those network cards are based on the NFP-4xxx silicon, a many-core architecture with 72 programmable cores with 8 threads each, and 2GB DRAM for lookup and state tables. A proprietary P4 compiler from Netronome was used for the evaluation.

The cards which were used have two ports supporting 10 Gbit/s Ethernet. In order to provide comparable results with the previous target with four ports, two cards were used. To enable communication between the two cards, frames need to be copied from one card to the other. A simple DPDK program without any packet processing logic running on the local CPU of the test platform was used for this purpose. As illustrated in Figure 3a, frames coming from one NPU (Network Processing Unit) are first processed on board, then copied to the main memory for forwarding by the CPU, and finally copied to the second NPU. In order to minimize the latency jitter, all packets are always sent to the CPU, even if the final destination is on the same NPU and could be forwarded without CPU involvement.

While the Agilio CX SmartNIC requires a server platform and as such is not suited for the integration into an embedded environment, the NFP-4xxx silicon on which it is based, might be. Therefore, to additionally evaluate the performance of the chip, a second series of measurements was performed without using the datapath through the CPU, assuming that the performance of the SmartNIC in this setup is mainly constrained by the NFP silicon. The packets were directly forwarded back to back from one physical port to the other, as illustrated in Figure 3b. To get comparable results, the same P4 program was used in both cases with different table configurations programming the destination of each packet.



(a) With CPU processing (b) Without CPU processing

Figure 3: Data paths used in the network processor platform

3) Prototype FPGA based target: The FPGA target is based on the Xilinx Zynq-7035 MPSoC [26]. It features a dual core ARM Cortex-A9 processor system (PS) which is closely coupled to a programmable logic (PL) based on the Kintex architecture with 275k logic cells and, in our configuration, eight high speed serial "GTX" transceivers, able to operate

at up to 10.3125 GHz bit rate. Five of these transceivers are used as 10GBASE-R Ethernet ports. The main part of the firmware is an Ethernet switch with optional TSN (Time Sensitive Networking) features. The switch is connected to the external 10GBASE-R ports, as well as to internal virtual network interface cards (VNIC) connected to the CPU (as illustrated in Figure 4).

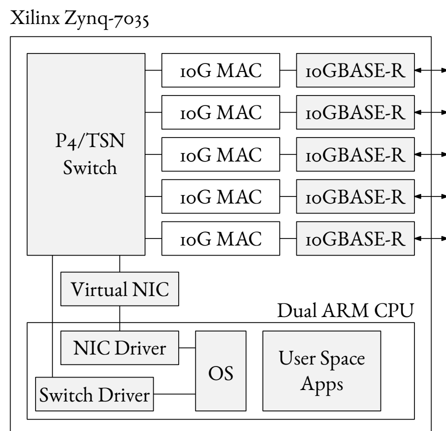


Figure 4: Overview of the implemented Zynq firmware

The packet processing and forwarding is performed completely by the switch core within the PL without any involvement of the PS, allowing maximum throughput and minimum latency. The architecture of the switch itself is outlined in Figure 5. The incoming Ethernet packets from each port (external or internal) are queued to ingress buffers from which they are multiplexed in a round-robin manner to the central processing pipeline. There, the MAC header information are extracted and looked up in hashed content addressable memories (CAMs) to determine the destination port(s) and queues. A demultiplexer distributes the packets to the selected queues and seamlessly duplicates the packets for multicasting. If the selected queues (implemented as dedicated block RAMs) are full, the packets may be buffered in external DDR3 memory.

Each port has 8 egress queues. The order in which the non-empty queues are selected for transmission is determined by a combination of strict priority [27, Sec 8.6.8.1], round robin and the TSN algorithms CBS (Credit Based Shaper) [27, Sec 8.6.8.2] and TAS (Time Aware Shaper) [28, Sec 8.6.8.4]. The parameters for the different algorithms are run time configurable per queue. For the following measurements, only one queue was used per port without any scheduling or traffic shaping.

In the current implementation, the packet processing logic is hardcoded in VHDL but is already prepared to be substituted by P4 generated cores to enable fast and flexible modifications to the behavior. For that, we are currently aiming at two very promising approaches. Xilinx has included a P4 compiler into its SDNet Development Environment [29] which translates the P4 description into an IP core which can be integrated to the FPGA firmware. A drawback might be the restriction to Xilinx FPGAs. The second approach comes from Netcope Technologies which offers a P4 to VHDL compiler [30] together with a suite of networking IP cores. In either approach, the generated packet processing pipeline has a fixed function completely described by the P4 program which is a very important aspect with regard to safety assessment and certification. Unfortunately, at the time of writing none of the two approaches were ready for implementation into our firmware.

It's further to be noted, that the switch core was originally designed for a 1 Gbit/s switch and therefore currently has a limited internal bandwidth of around 34 Gbit/s which obviously is not sufficient to serve five 10 Gbit/s interfaces at full line rate. It's therefore possible for the ingress buffers to overflow causing packets to be dropped at ingress before processing and before assigning priorities.

B. Measurement setup

Our measurement setup is presented in Figure 6. Traffic was generated by an Anritsu Network Analyzer MD1230B [31] on four different Ethernet links, generating traffic for a utilization from 0 % to 100 % (ie. up to 4 Gbit/s for the software based platform, and 40 Gbit/s for the NPU and FPGA platforms). This traffic is then processed and forwarded to the according output ports on the target platform.

Regarding the P4 program which was used for making the measurements, we used here a simple layer 2 Ethernet switch. This P4 program parses the Ethernet header and decides which output port(s) to use based on a learned or statically, run-time configured MAC address table.

C. Framerate

Figure 7 presents the framerate achieved by the three platforms for different packet sizes. Note that an ideal platform would be able to forward 100 % of the workload for the transmitted framerate presented in Figure 7. For packet sizes of 64 B, the software-based platform reaches a bottleneck at around 2.2 Mpps or 1.1 Gbit/s, meaning that it is only able to process 28 % of the full workload of 4 Gbit/s.

In case of the network processor platform, the bottleneck is reached at 8.7 Gbit/s with the CPU and 23 Gbit/s without CPU, meaning it is able to process respectively 21.8 % and 57.5 % of the full workload of 40 Gbit/s. The difference between the two measurements is due to the direct forwarding of all packets without any involvement of the CPU or main memory, which is also the bottleneck for the software platform.

The FPGA platform is only able to process a maximum of 5.6 Gbit/s with 64 B packet size. Using 1518 B packets, the maximum throughput of 28 Gbit/s is closer to the internal bandwidth limit of 34 Gbit/s. In addition to the internal bandwidth, overhead in the processing pipeline (mainly waiting states) limits the throughput of the FPGA target. Again, it's to be noted, that the switch in use was originally designed for 1 Gbit/s line rate only and further optimizations are necessary for this configuration.

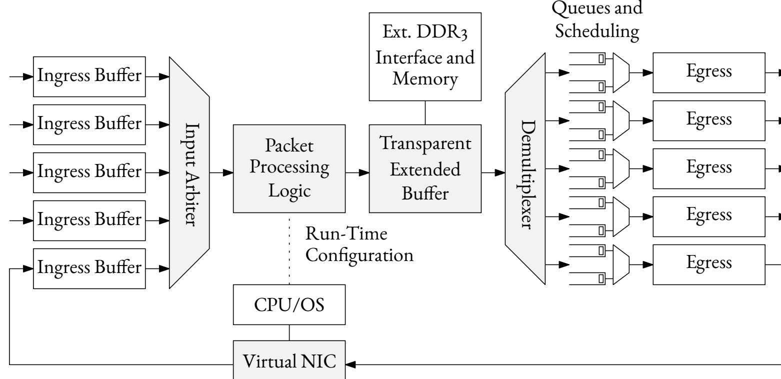


Figure 5: Simplified outline of the FPGA switch core

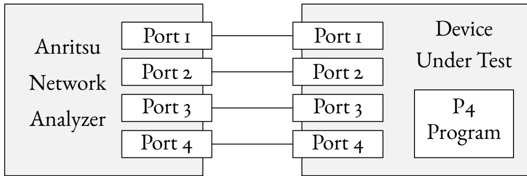


Figure 6: Measurement setup used for the performance evaluation. Link speeds were either 1 Gbit/s or 10 Gbit/s

D. Packet processing latency

Figure 8 presents the packet processing latency as a function of the time between two frames (or framegap). We notice that for framegaps larger than 1 µs the processing latency is of 24 µs for packet sizes of 1518 B for the software-based and network processor platforms. Since both approaches require copies of the frames from the network cards to the CPU, similar latencies are expected. For framegaps smaller than 1 µs the processing latency of the software-based platform increases up to 1 ms depending on the packet size. The packet processing is not able to keep up with the incoming rate and packets are buffered leading to the increased latency. Once the buffers are full, packets will be dropped, as already shown in Figure 7.

The network processor platform without CPU is able to better cope with the more intensive traffic, which can be explained by the fact that the processing is completely performed by the NPU, without CPU involvement or the need to copy packets.

The FPGA-based platform produces the best latencies, with values around 1.2 µs without buffering (i.e. for large framegaps). Once the internal bandwidth limit is hit, packets are buffered at ingress and eventually dropped. The latencies up to 15.8 µs observed in this region of small framegaps correspond to the capacity of the ingress buffers, which are much smaller compared to the software/network processor implementations and thus leading to smaller latencies in these situations, but potentially more packet losses during short, intense traffic bursts. However, in network systems with hard real-time requirements, buffering of packets is unintended in order to keep the worst-case latency short and predictable.

A comparison between the measured values on the three platforms and previous work [11] done on an industrial AFDX switch and an HP E3800 switch is presented in Table I. The gap in processing latencies for software-based P4 platforms compared to purely hardware-based ones still makes those platforms attractive for use-cases where latency requirements are less strict.

Switch	**Proc. latency**
Rockwell Collins AFDX switch	5 µs
HP E3800 without OpenFlow	7.2 µs
HP E3800 with OpenFlow	7.7 µs
HP E3800 with software switching	613 µs *(avg.)*
(Section V-A1) P4 software switch with DPDK	24 µs
(Section V-A2) P4 switch with NPU and CPU	24 µs
(Section V-A2) P4 switch with NPU and w/o CPU	5.8 µs
(Section V-A3) Switch with FPGA platform	1.2 µs

Table I: Comparison of packet processing latency of the evaluated P4 switch with numerical results from [11]

E. *Profiling of software-based target*

Software profiling was performed in order to better understand the software-based platform from Section V-A1. Figure 9 presents the profiling of the P4 program using the `operf`[5] statistical profiler for Linux. This tool enables us to evaluate how much time is spent in each function of the P4 program and the system. Note that only 2 cores of the CPU were used for this measurement, explaining the maximum value of 200 %. Three different function groups are presented in Figure 9:

- P4 primitives: *Parser*, *Table* and *Actions*, which is the part taking the most resources (up to 70 %);
- DPDK primitives: *Ethernet Driver*, *Ethernet Library* and *Run Time Environment (RTE)*, which take relatively low resources compared to the P4 primitives;
- Other functions: the C standard library (`libc`), Linux kernel, and overhead.

[5]http://oprofile.sourceforge.net

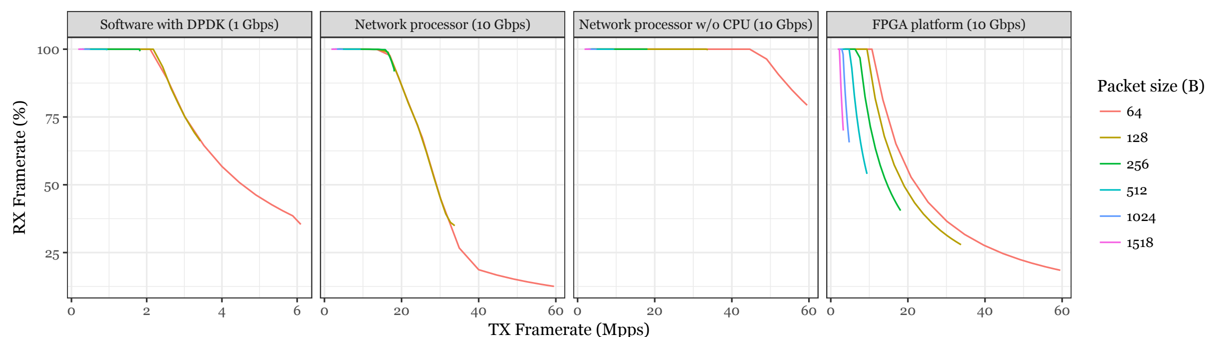


Figure 7: Framerate processed by the P4 switch

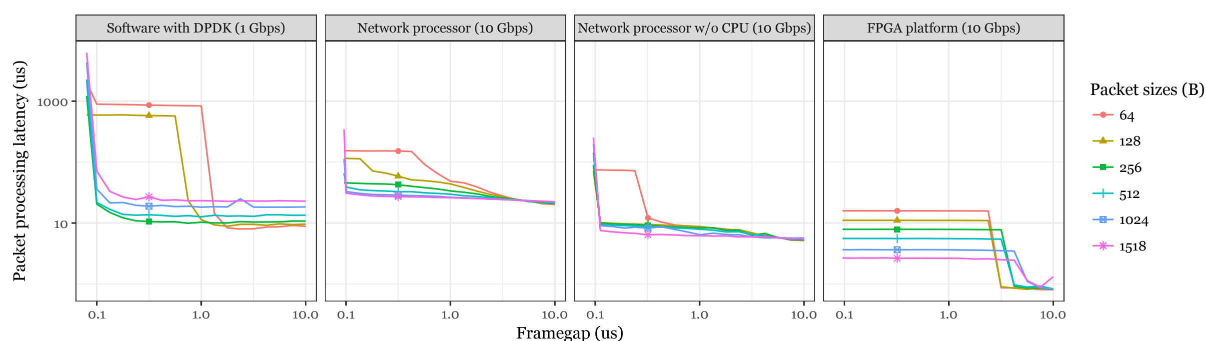


Figure 8: Packet processing latency as a function of the time between two frames

While in the tested setup the P4 program is able to almost fully process the 4 Gbit/s of traffic as shown in Section V-C, some work on reducing the P4 resources must be done in case additional ports would be used.

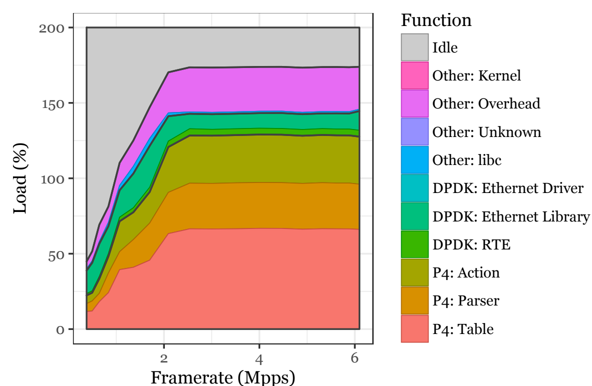


Figure 9: Profiling of the Linux computer with 64 B packet sizes

VI. Conclusion

We investigated in this paper recent developments made in the area of network-specific reconfigurable hardware and associated description language, namely the P4 programing language. This new development allows faster development of customized packet processing devices such as Ethernet switches or routers without the need for a deep knowledge about the target hardware architecture.

We presented in this paper P4 and its functionalities. We showed that its high flexibility in combination with simple building block enabling a formal analysis make it an attractive platform for some network protocols used in aeronautical use-cases. While some features such as definition of advanced egress packet scheduling and methods for time-based or time-triggered protocols are still lacking for more advanced network protocols, recent additions to the language in $P4_{16}$ and the Portable Switch Architecture make it an attractive platform. A case study of implementing AFDX was also performed in order to demonstrate that aeronautical protocols may be implemented using P4.

A performance evaluation of a simple P4 program was carried out on three different platforms: purely software-based target based on Intel DPDK, a hardware network accelerator based on a Network Processor Unit, and a FPGA-based platform. A comparison with an aeronautical and a COTS switch showed that while hardware based platform outperform software-based solutions on processing latencies, the difference between software and hardware solutions would be acceptable in some applications.

Since P4 is still under active development, with changes

adding incompatibility between its different version, it is not yet ready for production use in aeronautical use-cases with long lifetimes. Nevertheless, initiatives such as the Portable Switch Architectures lead the way to standardized capabilities, meaning more stability in the future. Such platform is of high importance in the area of prototyping where functional validation and some indication about performance evaluation are necessary. Its simple cost model and associated formal analysis also make it a good target for future certification of packet processing devices.

Future work will include more in-depth studies of P4 and its platforms, with possible evaluations and deployment in services where frequent update is necessary, such as passenger connectivity. Further, TSN techniques like stream reservation and bandwidth allocation using the dedicated scheduling algorithms in conjunction with P4 will be investigated. Another area of interest would be examining methods, models and associated tools for certification of such approaches.

Acknowledgements

The authors would like to thank AED Engineering GmbH for providing access to the FPGA switch core and Netronome as well as its Open-NFP community for their support and fruitful discussions on the Agilio CX SmartNIC.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[2] Intel, "Intel DPDK: Data Plane Development Kit," Accessed 2021/02/27. [Online]. Available: http://dpdk.org

[3] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[4] G. Brebner and W. Jiang, "High-Speed Packet Processing using Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, Jan. 2014.

[5] R. Giladi, *Network Processors: Architecture, Programming, and Implementation*. Morgan Kaufmann, 2008.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013, pp. 99–110.

[7] H. Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in Software Defined Networking*, 2013, pp. 127–132.

[8] M. Dobrescu, K. 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 12)*, Apr. 2012, pp. 141–154.

[9] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems," *Proceedings of the 14th International Conference on Networks (ICN 2015)*, pp. 78–83, Apr. 2015.

[10] D. Henneke, L. Wisniewski, and J. Jasperneite, "Analysis of Realizing a Future Industrial Network by Means of Software-Defined Networking (SDN)," in *12th IEEE World Conference on Factory Communication Systems (WFCS 2016), Aveiro, Portugal*, Aveiro, Portugal, May 2016.

[13] ——, "P4$_{16}$ Language Specification," Version 1.0.0, May 2017. [Online]. Available: http://p4.org/spec/

[11] P. Heise, F. Geyer, and R. Obermaisser, "Deterministic OpenFlow: Performance Evaluation of SDN Hardware for Avionic Networks," in *Proceedings of the 11th International Conference on Network and Service Management (CNSM)*, Nov. 2015, pp. 372–377.

[12] The P4 Language Consortium, "The P4 Language Specification," Version 1.0.4, May 2017. [Online]. Available: http://p4.org/spec/

[14] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. ACM, Oct. 2010, pp. 19:1–19:6.

[15] The P4 Language Consortium, "P4→NetFPGA: A low-cost solution for testing P4 programs in hardware," Accessed 2021/02/27. [Online]. Available: https://p4.org/p4/p4-netfpga-a-low-cost-solution-for-testing-p4-programs-in-hardware/

[16] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, Aug. 1989.

[17] M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round-Robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, Jun. 1996.

[18] F. Geyer, S. Schneele, M. Heinisch, and P. Klose, "Simulation and Performance Evaluation of an Aircraft Cabin Network Node," in *Proceedings of the 4th International Workshop on Aircraft System Technologies*, ser. AST 2013, F. T. Otto von Estorff, Ed. Shaker-Verlag, Apr. 2013, pp. 323–332.

[19] The P4 Language Consortium, "P4$_{16}$ Portable Switch Architecture (PSA)," (draft), May 2017. [Online]. Available: http://p4.org/spec/

[20] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable Packet Scheduling at Line Rate," in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016, pp. 44–57.

[21] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet Transactions: High-Level Programming for Line-Rate Switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, pp. 15–28.

[22] Aeronautical Radio Inc., "ARINC Specification 664P7-1: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network," Sep. 2009.

[23] J. Heinanen and R. Guerin, "A Two Rate Three Color Marker," RFC 2698 (Informational), Internet Engineering Task Force, Sep. 1999.

[24] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel, "High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 629–630.

[25] Netronome Systems, Inc., "Product Brief: Agilio® CX 2x10GbE SmartNIC," 2017. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[26] Xilinx Inc, "Zynq-7000 All Programmable SoC," 2017. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[27] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks," *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, pp. 1–1832, Dec 2014.

[28] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q—as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)*, pp. 1–57, March 2016.

[29] Xilinx Inc, "SDNet Development Environment," 2017. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

[30] Netcope Technologies, a.s., "Netcope - P4 to VHDL," 2017. [Online]. Available: https://www.netcope.com/en/products/p4-to-vhdl

[31] Anritsu, "Data Quality Analyzer MD1230B," Accessed 2021/02/27. [Online]. Available: http://www.anritsu.com/en-US/test-measurement/products/MD1230B

A.3.4 Rapid Prototyping of Avionic Applications Using P4

This work was published in *5th P4 Workshop*, 2018 [155].

Rapid Prototyping of Avionic Applications Using P4

Dominik Scholz, Fabien Geyer, Sebastian Gallenmüller, and Georg Carle
Chair of Network Architectures and Services, Department of Informatics, Technical University of Munich, Germany
Email: {scholz, fgeyer, gallenmu, carle}@net.in.tum.de

I. Applicability for Aeronautical Applications

The main advantage of P4 is the decorrelation between the behavior of a packet processing device and the hardware which is used. It means that engineers are not tied to a specific set of network protocols implemented by hardware vendors. This is especially relevant in the aeronautical industry where specific network protocols (e.g. ARINC standards) are used. Furthermore, devices must exclusively implement the required protocols and functionalities for safety reasons. Those two constraints prevent commodity devices to be used since they either do not support the required protocols, or implement a larger set of protocols than the ones required. With P4, the usability of commodity devices increases.

Another advantage of P4 is the simplicity and constraints put on the abstract forwarding model. Since P4 forbids dynamic memory allocation and iterations with unknown counts, formal derivations of worst-case execution time and resource usage of a P4 program are straightforward already at compile time. This is again relevant in the aeronautical industry since constraints on those cost factors are required in real-time applications.

A downside when using P4 for aeronautical applications is that egress packet scheduling cannot be directly described by P4. This limits the description of more advanced Quality-of-Service architectures envisioned for next-generation aeronautical backbones [1].

Since safety is an important aspect of aeronautical applications, specification and programing languages need to have defined behavior. While this was a problem in the 2014 specification of P4 as some aspects are incompletely specified (e.g. casting between different data types, and initial values of table entries and packet attributes) this has since been solved with $P4_{16}$.

Finally, time-based or time-triggered protocols cannot be directly described using P4 since there are no primitives for describing access to a clocking information, besides timestamps at ingress and egress in $P4_{16}$. This drawback prevents the implementation of time-synchronization protocols for packet timestamping, or egress scheduling based on time information.

II. Case Study

In order to evaluate the applicability of P4 to aeronautical use-cases, we applied P4 to the case-study of Avionics Full-Duplex Switched Ethernet (AFDX), an Ethernet-based protocol for safety-critical applications standardized in ARINC 664 Part 7 [2].

An AFDX network is composed of *end-systems* and *switches* as nodes. End-systems serve as source and destination nodes in the network, over which applications send data according to bandwidth restrictions to avoid overloading. One fundamental building block of AFDX is the notion of *virtual links* (VL), which can be seen as rate-constrained network tunnels. The parameters describing a VL are: the emitter end-system of this VL, the list of receiving end-systems, static routes between emitter and receivers, the Bandwidth Allocation Gap (BAG), as well as minimum and maximum frame length. The BAG is defined as the minimum time interval between the first bit of two consecutive frames from the same VL.

III. Implementation Overview

AFDX switches must ensure the following functionalities for each frame entering the switch: identification of the VL; frame filtering and policing based on VL parameters (allowed input port, BAG, and frame length limits); forwarding of the frame to the correct output ports based on the VL routing configuration.

Frames in AFDX are based on the standard Ethernet frame format. The VL identification number is encoded in the 16 least significant bits of the destination MAC address. Switch configuration and routes in P4 are saved in tables. For our case-study, we define a table containing the allowed input port of each VL and its output port. Incoming packets with invalid Virtual Link identifiers are dropped here.

Our case-study contains a simplified ingress function for AFDX with basic frame integrity checking, application of the aforementioned table, and policing. For the latter, a simple process based on validating the BAG timing properties and minimum and maximum frame sizes is described in the ARINC standard. While frame size validation is possible with P4, filtering based on the time between frames is not. Since from a functional point of view the goal is to limit the bandwidth of each VL, the standard policing mechanisms provided by P4 using meters may be used as an alternative.

Finally, the last step is to forward frames to the correct output ports. This is done using multicast by a vendor-specific mechanism.

IV. Further Aspects

Our implementation is based on $P4_{14}$ as by the time of implementation more hardware supported this P4 version. We listed here only a subset of the functionalities needed by an AFDX switch. More advanced features such as operational modes and monitoring functionalities based on SNMP can also be implemented using P4. We performed a performance analysis regarding throughput, latency and profiling for three P4 targets: An Intel DPDK-based software switch, a Netronome Agilio CX SmartNIC network processor platform, and a prototype FPGA-based platform.

Acknowledgments

This work was supported by the German-French Academy for the Industry of the Future. We thank Max Winkel for his valuable contributions. A full paper was published at ERTS 2018 [3].

References

[1] F. Geyer, S. Schneele, M. Heinisch, and P. Klose, "Simulation and Performance Evaluation of an Aircraft Cabin Network Node," in *Proceedings of the 4th International Workshop on Aircraft System Technologies*, ser. AST 2013, O. von Estorff and F. Thielecke, Eds. Shaker-Verlag, Apr. 2013.

[2] Aeronautical Radio Inc., "ARINC Specification 664P7-1: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network," Sep. 2009.

[3] F. Geyer and M. Winkel, "Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Jan. 2018.

Dominik Scholz is a Ph.D. student at the Chair for Network Architectures and Services at the Technical University of Munich. He received his M.Sc. in Informatics at the Technical University of Munich in 2016. His research interests include P4, high-performance software packet processing architectures, and reproducible performance measurements of network devices.

A.3.5 Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning

This work was published in *Proceedings of the 7th IEEE International Conference on Network Softwarization*, 2021 [139].

Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning

Peter Okelmann*, Leonardo Linguaglossa†, Fabien Geyer*, Paul Emmerich*, Georg Carle*
*Technical University of Munich †Telecom Paris
okelmann@in.tum.de, {fgeyer,emmericp,carle}@net.in.tum.de linguaglossa@telecom-paristech.fr

***Abstract*—Processing packets in batches is a common technique in high-speed software routers to improve routing efficiency and increase throughput. With the growing popularity of novel paradigms such as Network Function Virtualization, advocating for the replacement of hardware-based networking modules towards software-based network functions deployed on commodity servers, we observe that batching techniques have been successfully implemented to reduce the HW/SW performance gap. As batch creation and management is at the very core of high-speed packet processors, it provides a significant impact to the overall packet processing capabilities of the system, affecting latency, throughput, CPU utilization and power consumption. It is commonly accepted to adopt a fixed maximum batching size (usually in the range between 32 and 512) to optimize for the worst case scenario (i.e. minimum-size packets at full bandwidth capacity). Such approach may result in a loss of efficiency despite a 100% utilization of the CPU. In this work we explore the possibilities of enhancing the runtime batch creation in VPP, a popular software router based on the Intel DPDK framework. Instead of relying on the automatic batch creation, we apply machine learning techniques to optimize the batching size for lower CPU-time and higher power efficiency in average scenarios, while maintaining its high performance in the worst case.**

I. Introduction

Software packet processing has become a commonly adopted alternative to costly, expensive hardware-based processing engines [1], [2]. Together with the inherent flexibility advantages of software-based solutions w.r.t. hardware counterparts, a current trend shows that the performance gap is also diminishing [3]. As a consequence, several libraries for high-speed packet processing on pure software such as the Intel's DPDK[1] or Netmap [4] are being used as building blocks for a flourishing ecosystem of software middleboxes capable of performing multi-10-Gbps packet processing on a single CPU core of commercial off-the-shelf (COTS) servers.

Modern tools for software packet processing, also known as *software routers*, incorporate many optimizations such as processing packets in batches and adopting a kernel-bypass approach to access the Network Interface Cards (NICs) with pure user-space drivers and minimize the interference of low-level system calls by the operating system. In particular, batching is usually adopted in conjunction with a busy polling behavior: the CPU continuously performs a loop to verify if any packet is received at the NIC, then it uses a minimalistic batch creation algorithm to process a full batch of packets (as opposed to per-packet processing) and it repeats the loop at the end of the processing. Batching and busy polling are very effective in high-load scenarios, where the cost of interrupt handling per packet could saturate the CPU. In particular, it has been shown that increasing the batch size positively correlates with a significant improvement in the packet processing rate, up to a certain saturation threshold [5], [1]. Therefore, the achievable throughput is maximized at the cost of a 100% CPU utilization even in the case of low-load scenarios, resulting in a lot of wasted CPU cycles and high power consumption.

While the maximum batch size is fixed (usually 32 to 512), the actual size depends on the number of packets waiting in the NIC's input queues. But the actual batch size also affects the processing efficiency, with small batches requiring more clock cycles per packet than large ones. This causes a feedback loop, where oscillating batch size can be observed in scenarios where the input load does not fully saturate the CPU. Such a batching approach provides opportunities for improvement: ideally, in a non-saturated regime (i.e., no packet loss) the CPU can be relieved of some processing if we keep the batch sizes large enough to maintain the processing efficiency.

In this paper[2], we propose an algorithm to dynamically allocate batch sizes depending on the traffic condition instead of the classical busy polling approaches. With the help of a large dataset collected over hours of experiments with a real packet processing engine, we first develop machine learning techniques to find the optimal batching size for different load scenarios. We then deploy our training model within a software router, and assess the impact of our approach in terms of saved clock cycles. The remainder of the manuscript is organized as follows: Section II provides a background on the relevant architectural aspects of the software router of our choice (namely, VPP [6]) and the related work. In Section III we analyze the possible improvements for the batching algorithm and present our design space. We then evaluate our system in Section IV and Section V concludes the manuscript.

II. Background and related work

In a softwarized network scenario, a *software router* is a piece of code running on a general-purpose server which is responsible for moving packets from one NIC (or more) to a network application for further processing. Since NICs can be both physical or virtual, software routers are fundamental components of virtualized network systems. When a NIC is

[1] `https://www.dpdk.org/`
[2] Our source code and scripts: https://github.com/pogobanane/vpp-testing

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

equipped with multiple hardware queues, a software router process is usually bound to a single queue and executed by a single CPU, to allow horizontal scalability while, at the same time, avoiding inter-process interruptions [7]. The majority of high-speed software routers can be executed within Linux environments, and make extensive usage of low-level libraries such as Intel's DPDK or netmap [4] rather than relying on the standard libraries for packet processing. Such libraries are optimized to maximize the computational efficiency of the packet processing application, and provide the additional advantage of avoiding the overhead of the Linux kernel [1].

A. *Vector packet processing*

As a use case for our work, we select Vector Packet Processing (VPP), a high speed packet processor originally developed by Cisco and recently released as an open-source Linux Foundation's project named FD.io [6]. VPP provides a rich feature set for a wide range of hardware and architectures, and adopts most of the popular design choices to improve the packet processing rate [1]. To improve modularity and ease of programming, most of VPP's features are developed as individual plugins, that are further organized as nodes in a *processing graph*, which represents the desired packet processing applications. When a physical NIC is controlled by a DPDK driver, the first node accessed upon packet reception is the `dpdk-input` node, which is responsible for querying packets from the NIC via the DPDK library, creating a batch with the received packets and passing the full batch to the next node. Subsequent nodes can differ, depending on the required network stack to be accessed. For example, in the case of IPv4 packets, a following node may be the `ip4-input` that will parse the IPv4 headers, or a `ip6-input` that will deal with IPv6 packets. At the end of the processing, a final output decision is taken at the `dpdk-output` node which can choose to forward packets to another physical or virtual NIC.

With the adoption of receive-side scaling (RSS), VPP's main thread can distribute the incoming traffic to multiple worker threads. Each thread then runs its own instance of the processing graph. Worker threads can be conveniently pinned to cores and assigned a scheduler via VPP's configuration. This allows reaching higher throughput when processing multiple traffic flows. Since we are interested in changing the batch creation behavior, we focus our investigation on the `dpdk-input` node.

B. *Batch creation and CPU behavior*

The unmodified version of `dpdk-input` implements a busy-loop which continuously polls new packets from the NIC using the call `rte_eth_rx_burst`. If no packets have arrived, it simply returns to the beginning of the loop, which leads to the same node being called again. In this way, while the main thread is busy waiting for new packets, the CPU is continuously utilized at 100%. If a poll detects some packets queued at the NIC, the DPDK library tries to retrieve as many packets as possible until the maximum batch size has been reached (defaults to a value of 256). It is worth noting that when the DPDK node detects less than 32 packets waiting at the NIC,

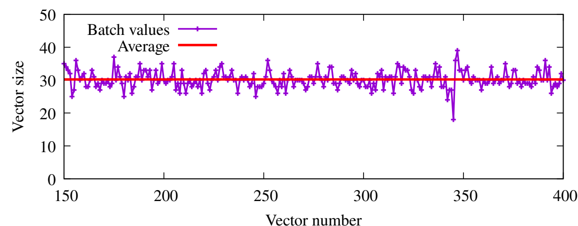


Figure 1: Oscillation of batch sizes for a L2 forwarding function that processes a constant 5 Gbit/s traffic.

the loop returns immediately as the driver assumes that no more packets will be arriving at the queue. This proves useful in low-load scenarios, as packets are immediately batched and submitted into the processing graph to keep latency low.

The size of soon-to-be-processed batches highly affects the computational efficiency of the underlying CPU. Considering a constant-bit-rate (CBR) scenario, when a large batch is received the CPU efficiency is very high [1], and as a result, the processing time per packet is low. Since the polling loop can quickly return to retrieve more packets, the next poll will retrieve less packets because of the constant bit rate. A smaller batch, will result in a lower efficiency and, therefore, a higher processing time per packet. This will in turn result in more packets being queued at the NIC, and another new poll with a larger batch size. This oscillating behavior can be observed in Figure 1, which shows the number of packets in a batch as a function of the time, for an average load of 5 Gbit/s. This behavior leads the CPU to switch between higher power-consumption condition, back to low-energy consumption. Moreover, batch sizes also have an impact on latencies of the packets [8]. Keeping in mind that the CPU is continuously utilized at 100 %, we propose a different approach which tries to (i) minimize the oscillating behavior, (ii) keep the CPU efficiency always at its maximum and (iii) release the CPU occupancy by using an idle state which will free some clock cycles (that can be used by other concurrent applications).

C. *Related work*

Optimizations of batching behavior are closely related to our work: for example, SmartBatching [8] aims at adapting the batching behavior according to an analytical model derived from the input load, which improves both CPU behavior and latency. Similarly, Metronome [9] is an approach to replace the continuous polling with a sleep and wake intermittent mode and an optimized CPU `sleep` function. Analytical modeling is becoming widely adopted to provide previsions on key performance metrics such as the packet loss or the expected batch sizes, as done in [10]. Our work differs from the previous in that it relies on machine learning to adapt to the incoming input rate, rather than on analytical modeling. This way, we can relax the assumptions on the input traffic pattern, as we just need to train our model with realistic traffic. We rely on the standard Linux `nanosleep` without any additional kernel module. A different approach is adopted by Shenango [11],

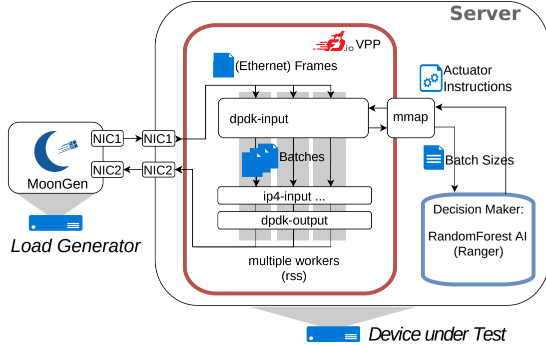


Figure 2: The testbed used for our experimental evaluation, which includes the MoonGen traffic generator, the device under test and the AI component.

which divides the available CPU cycles into fine-grain slots, while a separate orchestrator reallocates the CPU cores and steers the input traffic depending on latency requirements. Our method is more flexible in that we do not require a fixed time-slot for the reallocation of CPU cycles.

III. Design Space

As explained in Section II, the batch size used by VPP depends on the load at the NIC. Although VPP tries to create maximum-size batches in high-load, it may be beneficial to find a different value in lower load scenarios. For example it can be useful to keep batches artificially large while, at the same time, saving some CPU clock cycles for further processing. In the scenario depicted in Figure 1 this may help to reduce the batch size oscillations as well as freeing some CPU clock cycles which can be used for different processing. In a virtualized environment, CPU cores may be shared with other applications. With a tuned value for the batch sizes, the `dpdk-input` node could then free the CPU core for other applications. This can be done with the UNIX `nanosleep` syscall. Another simple but effective measure could be to scale the frequency of the CPU clock in order to reduce the power consumption and reduce the clock cycles allocated for the busy-polling. Finally, it can be possible to enable the interrupt mode for very low load scenarios, though this approach would require a fine tuning of the switching threshold. Table I summarizes the possible actions. We now show how to estimate the optimal values for the batch sizes.

A. Decision making via ML

The aforementioned modifications are beneficial only in certain scenarios. Their usage must be adapted to the current load situation by a *decision maker* (DM). This process can be defined through modeling, which however is prone to errors as it is hard to create a representation of the system that is sufficiently detailed to capture the low-level details that are essential for our processing [10]. As the actions influence latency, throughput, CPU usage and power consumption, the decision making process must optimize for all of those. Additionally, most actions will also affect the subsequent state of the system, thus incurring in feedback actions and non-linear effects. We opt for a machine learning solution as an alternative to classical analytical approaches, as it can be used to approximate a solution for such a complex problem, without the necessity of manually modeling and tuning the system model. The DM uses a list of the last batch sizes as input for its actions. In combination with the parameters to optimize for and the possible actions to tune, this results in a problem of high dimensionality. Our ML decision maker is used to regularly update the threashold configuration for using actions.

B. Architecture overview

We now describe the proposed architecture, as shown in Figure 2. The device under test (DUT) consists of two parallel components: the software router, and the decision maker. Every time the `dpdk-input` node submits a batch of packets to the processing graph, it also communicates the batch size to the DM. The ML algorithm then runs its predictions and returns the new, updated action instructions which are in turn read by VPP. For the IPC communication we adopt non-blocking I/O in order to keep high throughput performance.

For the DM component, it is essential to use fast ML techniques, as otherwise the efficiency advantages would be negated by the resource hungry machine learning component. We selected random forests and ranger [12], as they are efficient and easy to integrate in VPP.

In theory, it is possible to alter the state of the system by adopting a combination of all the actions shown in Table I, depending on the severity of the impact on the processing. For example, switching to interrupt mode would reduce the load on the CPU, at the cost of a severe performance degradation. However, the base Ranger version comes with a limited interface with no support for multi-dimensional variables. Therefore the batch selection must be controlled by a single variable (and thus, a single action can be used). Although all the mechanisms shown in Table I are implemented, we focus here on `nanosleep` actions controlled by a single integer.

Description	Implemented	Used
release the CPU (`nanosleep`)	✓	✓
delay the polling (`rte_delay_us`)	✓	✗
interrupt mode (~~`rte_eth_dev_rx_intr_*`~~)	✓*	✗
CPU freq hints (`rte_power_freq_up`)	✗	✗

Table I: Proposed actions and their usage by ranger.
* : Implemented, but not functional.

C. Random Forest Training

The ranger API is used for training a forest taking the latest used batch sizes of VPP as input, and to predict the best time to nanosleep for. As presented in Figure 2, VPP and ranger run on the DuT, while load scenarios are performed by the load generator running MoonGen [13]. White box measurements like clock cycle counting are conducted on the DuT, and black box measurements like latency and throughput are collected by the load generator.

Method	msg/s	avg (µs)	min (µs)	max (µs)	std dev (µs)
mmap	867,092	1.103	1.024	5.376	0.178
shm	726,068	1.377	1.320	1.024	0.334
fifo	76,029	13.037	10.496	27.396	0.751
pipe	59,972	16.674	14.557	120.320	3.960

Table II: Comparison of IPC with 1000 messages of 4096 bytes each

In order to train the random forest, an iterative process is used. After each run of the performance measurements, the success of the forest is evaluated using a reward function. The reward is then used to refine the prediction values to train for which combined with the newly collected ranger input batch sizes make up the new training set. Finally, the next iteration of the forest can be trained and the next training iteration begins.

We limited our evaluation to six scenarios with different packet rates: 2, 10, 500, 1000, 5000 and 7500 Mbit/s. Since we focus on the `nanosleep` action, we selected low load scenarios where freeing CPU cores is beneficial, but also included larger loads to avoid over-fitting.

In each training iteration those six scenarios are run and used to refine the training set. Finding the correct prediction results relies on a reward function r. It weights the average latency l in µs, the average throughput t in packets per second and the CPU cycles c used by VPP as follows:

$$r(l,t,c) = -0.5l + \frac{4000t}{c * 10^{-7}}$$

Afterwards the range of the reward is limited to the range $[0,-1]$ using the expected minimum and maximum reward values $r_{min} = -10$ and $r_{max} = 120$:

$$p_{deviate}(r) = 1 - \frac{r}{|r_{min} - r_{max}|}$$

This is the probability with which the latest prediction p_{last} should be changed. Next the new prediction result to be trained for is drawn using a random function with a normal distribution, p_{last} as center and s as standard deviation. It is based on p_{last}, $p_{deviate}$ and a constant to guarantee the continued exploration of new values $c_{explore} = 5$:

$$s(p_{deviate}) = 100 * p_{deviate}^5 + c_{explore}$$

The more the reward r rises, the smaller becomes $p_{deviate}$ which in turn results in an aggressive reduction of s. Using that system over many training iterations, the random forest output is able to converge.

IV. Numerical evaluation

We numerically evaluate in this section the different components of our architecture. We also illustrate the impact that our machine-learning based software router has on the overall performance. Our benchmarks are performed on a server with an Intel Xeon CPU E31230 at 3.20 GHz.

The traffic generated by MoonGen consists of 64 B packets since it is most demanding scenario for the software component of software routers. Other parameters which were not tested, like packet size, can have an impact though as well on packet processing times and thereby also influence latency.

Stage of Integration	Throughput	Ratio
Unmodified VPP	14.15 Mpps	100 %
Logging only	13.95 Mpps	99 %
Logging + Exporting	13.94 Mpps	99 %
...+ Exporting + Ranger load	11.57 Mpps	82 %
...+ Final trained forest	12.26 Mpps	87 %

Table III: Maximum throughput at different stages integration.

A. Inter-Process Communication

We evaluate here the solution used for communicating batch sizes and instructions between VPP and ranger. VPP already has the `elog` system[3] for in place logging. While small logging events should be performant, previous works [14] show that it can significantly impact the performance. Using an open-source benchmarking suite[4], we evaluated alternative IPC channels (see Table II). Based on those results, we built a more efficient communication channel using mmap.

Table III summarizes the impact of data collection and export on VPP's throughput. Running VPP while logging all batch sizes into the shared memory, results in a maximum throughput of 99 %. Running ranger for a single prediction, meaning exporting the ringbuffer only once to ranger, results in a similar throughput performance of 99 %. When running the ringbuffer export and prediction in an endless loop, VPP's throughput drops to 11.57 Mpps which corresponds to 82 % of the performance of unmodified VPP. Compared to other approaches using `elog` [14], showing a performance loss of 2 to 3 times, our mmap-based IPC proved to be more efficient.

Scenario	Pred./s	Std. dev.	Batches/Pred.
Random data	22 106	168	2.6
Real IPC	712	46	82.3

Table IV: Ranger predictions rate

B. Ranger performance

We evaluate the prediction rate of ranger with the help of Table IV, which illustrates how many batches may be processed on a fully utilized 10 Gbit/s link per prediction. We first benchmark ranger with random data and our results showed 22 106 predictions/s in average. With real data on the IPC channel, the performance drops to only 712 predictions/s. The number of packet batches n_v processed between two ranger predictions can be calculated from the packet throughput t_r, the average size of a batch v_s and the prediction rate p_r: $n_v = t_r/(v_s * p_r)$

With $p_r = 22\,106$, we obtain 3 batches per prediction which is close to optimal. With $p_r = 712$, each prediction will be used for the next 82 batches. This performance gap could be improved with an optimized mechanism for the interaction between the AI and the DUT component.

[3] https://wiki.fd.io/view/VPP/elog

[4] https://github.com/goldsborough/ipc-bench

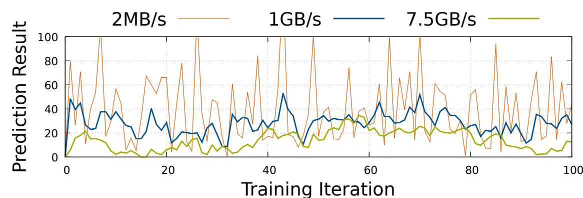


Figure 3: Convergence of the training in three scenarios.

C. Training

For each load scenario presented in Section III-C, we evaluated the prediction over the training iterations of the forest. Figure 3 shows that a convergence cannot be observed. For the higher loads of 1 Gbit/s and 7.5 Gbit/s, a trend emerges though: the random forest predicts higher nanosleep times for smaller load scenarios which is what a human expert would expect. For the smallest traffic rate used, the training could not converge. A reason could be that satisfying the reward function in situations with that little traffic is really hard because the CPU cycles per packet used by VPP seem to rise non-linearly for those scenarios. Our experiments show that the nanosleep time has to be several orders of magnitude higher than 50 µs to get to a CPU cycle efficiency expected by the reward function.

D. Validation and comparison of the system

Finally, we evaluate our architecture by measuring the CPU utilization in different scenarios and comparing it to an unmodified VPP (see Figure 4). Unmodified VPP has the worker thread running on CPU1. Its utilization is 100 % regardless of the offered load. The second core runs nothing and therefore has a utilization of 0 %. Depending on the traffic its (avg, max) latencies are between (5, 6)µs and (12, 65)µs.

Modified VPP with ranger updating the optimization instructions on CPU2 constantly utilizes about 98 %. The worker thread of VPP on CPU1 now shows a different behavior. When offered no load, the nanosleep time is set to 30 by the forest. This results in a utilization of only around 20 % on CPU1. When offered more load (1000 Mbit/s around time 20 s in Figure 4), the nanosleep time drops and CPU1 raises to 45 % load to process the packets. From offering 1200 Mbit/s of load onward (at time 31 s in Figure 4), VPP's worker thread starts hitting the upper limit of available cycles of CPU1. The latencies range from (12.5, 54)µs to (13.4, 101)µs. At a throughput of 12.26 Mpps the maximum performance of the system is finally found.

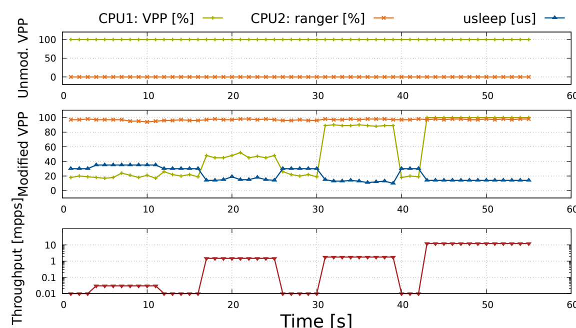


Figure 4: A comparison under the same load scenario. Unmodified: CPU1: VPP worker, CPU2: nothing. Modified: CPU1: VPP worker, CPU2: ranger.

V. Conclusion

We showed that the CPU utilization of VPP could be reduced in low load scenarios using random forests for finding optimization parameters at runtime. Regardless of the added context switches and complexity, the throughput performance in high load situations is reduced by only 13 %. Although a separate core is fully utilized by the ranger thread, future work could use the same core to save CPU time off VPP worker threads on multiple cores. Among possible optimizations, dynamically reducing ranger's refresh rate may be a promising one. Further research is also required for comparative study and evaluation of the effects on both throughput and latencies combined, because of the trade-off between CPU efficiency and processing times. Finally, exploring other random forest implementations may open possibilities for using more actions, improving training and implementing online learning to react to new, unknown traffic patterns.

References

[1] L. Linguaglossa, D. Rossi, S. Pontarelli, D. Barach, D. Marjon, and P. Pfister, "High-speed data plane and network functions virtualization by vectorizing packet processing," *Computer Networks*, 2019.

[2] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *Proc. of ACM/IEEE ANCS*, 2015.

[3] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proc. of the IEEE*, 2019.

[4] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *USENIX Security 12*, 2012.

[5] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. of ACM/IEEE ANCS*, 2015.

[6] "What is VPP?" https://wiki.fd.io/view/VPP/What_is_VPP%3F, accessed on 2020-12-01.

[7] T. Herbert and W. de Bruijn, "Scaling in the linux networking stack," https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011.

[8] M. Miao, W. Cheng, F. Ren, and J. Xie, "Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing," in *Proc of HPCC/SmartCity/DSS*.

[9] M. Faltelli, G. Belocchi, F. Quaglia, S. Pontarelli, and G. Bianchi, "Metronome: Adaptive and precise intermittent packet retrieval in DPDK," 2020.

[10] S. Lange, L. Linguaglossa, S. Geissler, D. Rossi, and T. Zinner, "Discrete-time modeling of NFV accelerators that exploit batched processing," in *IEEE INFOCOM 2019*, 2019.

[11] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX NSDI*, 2019.

[12] M. N. Wright and A. Ziegler, "ranger: A fast implementation of random forests for high dimensional data in C++ and R," *arXiv:1508.04409*, 2015.

[13] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *IMC*, 2015.

[14] L. Linguaglossa, F. Geyer, W. Shao, F. Brockners, and G. Carle, "Demonstrating the cost of collecting in-network measurements for high-speed VNFs," in *Proc. of TMA*, 2019.

A.4 List of publications

Publications of the author related to this habilitation thesis are subsequently listed in the chronological order of the date of their appearance.

2015

[72] **Fabien Geyer**, Alexandros Elefsiniotis, Dominic Schupke, and Stefan Schneele. Multi-Objective Optimization of Aircraft Networks for Weight, Performance and Survivability. In *Proceedings of the 7th International Workshop on Reliable Networks Design and Modeling*, RNDM 2015, pages 345–350, October 2015. ISBN 978-1-4673-8051-5. doi:10.1109/RNDM.2015.7325251

[81] Peter Heise, **Fabien Geyer**, and Roman Obermaisser. Deterministic OpenFlow: Performance Evaluation of SDN Hardware for Avionic Networks. In *Proceedings of the 11th International Conference on Network and Service Management*, CNSM 2015, pages 372–377, November 2015. doi:10.1109/CNSM.2015.7367385

2016

[66] **Fabien Geyer** and Georg Carle. Network Engineering for Real-Time Networks: Comparison of Automotive and Aeronautic Industries Approaches. *IEEE Communications Magazine*, 54(2):106–112, February 2016. ISSN 0163-6804. doi:10.1109/MCOM.2016.7402269

[83] Peter Heise, Marc Lasch, **Fabien Geyer**, and Roman Obermaisser. Self-Configuring Real-Time Communication Network based on OpenFlow. In *Proceedings of the 22nd IEEE International Symposium on Local and Metropolitan Area Networks*, LANMAN 2016, June 2016. doi:10.1109/LANMAN.2016.7548851

[67] **Fabien Geyer** and Georg Carle. Towards Automatic Performance Optimization of Networks Using Machine Learning. In *Proceedings of the 17th International Network Strategy and Planning Symposium*, NETWORKS 2016, September 2016. doi:10.1109/NETWKS.2016.7751147

[19] Steffen Bondorf and **Fabien Geyer**. Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows. In *Proceedings of the 10th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2016, October 2016. doi:10.4108/eai.25-10-2016.2266598

[82] Peter Heise, **Fabien Geyer**, and Roman Obermaisser. TSimNet: An industrial Time Sensitive Networking simulation framework based on OMNeT++. In *Proceedings of the 8th IFIP International Conference on New Technologies, Mobility and Security*, NTMS, November 2016. doi:10.1109/NTMS.2016.7792488

2017

[73] **Fabien Geyer**, Max Winkel, and Stefan Schneele. Rapid Prototyping of Packet Processing Devices for Aeronautical Applications. In Frank Thielecke Otto von Estorff, editor,

Proceedings of the 6th International Workshop on Aircraft System Technologies, AST 2017, pages 273–282. Shaker-Verlag, February 2017. ISBN 978-3-8440-5086-8

[59] **Fabien Geyer**. Routing Optimization for SDN Networks Based on Pivoting Rules for the Simplex Algorithm. In *Proceedings of the 13th International Conference on Design of Reliable Communication Networks*, DRCN 2017, pages 47–54, March 2017. ISBN 978-3-8007-4383-4

[84] Peter Heise, **Fabien Geyer**, and Roman Obermaisser. Self-Configuring Deterministic Network with In-Band Configuration Channel. In *Proceedings of the 4th IEEE International Conference on Software Defined Systems*, SDS 2017, May 2017. doi:10.1109/SDS.2017.7939158

[60] **Fabien Geyer**. Performance Evaluation of Network Topologies using Graph-Based Deep Learning. In *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2017, pages 20–27, December 2017. doi:10.1145/3150928.3150941

[196] Alireza Zamani, **Fabien Geyer**, Alexandros Elefsiniotis, and Anke Schmeink. Aircraft Network Deployment Optimization with k-Survivability. In *Proceedings of the 11th IEEE International Conference on Advanced Networks and Telecommunications Systems*, ANTS 2017. IEEE, December 2017. doi:10.1109/ANTS.2017.8384192

2018

[71] **Fabien Geyer** and Max Winkel. Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications. In *Proceedings of the 9th European Congress Embedded Real Time Software and Systems*, ERTS 2018, February 2018

[156] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, **Fabien Geyer**, and Georg Carle. Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking 2018*, May 2018. doi:10.23919/IFIPNetworking.2018.8696830

[155] Dominik Scholz, **Fabien Geyer**, Sebastian Gallenmüller, and Georg Carle. Rapid Prototyping of Avionic Applications Using P4. In *5th P4 Workshop*, Stanford, CA, USA, June 2018

[68] **Fabien Geyer** and Georg Carle. Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning. In *Proceedings of the 2018 SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, Big-DAMA 2018, pages 40–45, Budapest, Hungary, August 2018. doi:10.1145/3229607.3229610

[69] **Fabien Geyer** and Georg Carle. The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds. In *Proceedings of the 2018 International Workshop on Network Calculus and Applications*, NetCal 2018, pages 43–48, Vienna, Austria, September 2018. doi:10.1109/ITC30.2018.10060

[61] **Fabien Geyer**. DeepComNet: Performance Evaluation of Network Topologies using Graph-Based Deep Learning. *Performance Evaluation*, 130:1–16, April 2019. ISSN 0166-5316. doi:10.1016/j.peva.2018.12.003

2019

[20] Steffen Bondorf and **Fabien Geyer**. *Deterministic Network Calculus Analysis of Multicast Flows*, pages 63–78. Springer International Publishing, Cham, 2019. ISBN 978-3-319-92378-9. doi:10.1007/978-3-319-92378-9_5

[74] **Fabien Geyer**, Holger Kinkelin, Hendrik Leppelsack, Stefan Liebald, Dominik Scholz, Georg Carle, and Dominic Schupke. Performance Perspective on Private Distributed Ledger Technologies for Industrial Networks. In *Proceedings of the International Conference on Networked Systems*, NetSys 2019, March 2019. doi:10.1109/NetSys.2019.8854512

[62] **Fabien Geyer** and Steffen Bondorf. DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks. In *Proceedings of the 38th IEEE International Conference on Computer Communications*, INFOCOM 2019, April 2019. doi:10.1109/INFOCOM.2019.8737496

[70] **Fabien Geyer** and Stefan Schmid. DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning. In *IFIP Networking 2019*, May 2019. doi:10.23919/IFIPNetworking.2019.8816842

[92] Benedikt Jaeger, Dominik Scholz, Daniel Raumer, **Fabien Geyer**, and Georg Carle. Reproducible Measurements of TCP BBR Congestion Control. *Computer Communications*, May 2019. doi:10.1016/j.comcom.2019.05.011

[121] Leonardo Linguaglossa, **Fabien Geyer**, Wenqin Shao, Frank Brockners, and Georg Carle. Demonstrating the Cost of Collecting In-Network Measurements for High-Speed VNFs. In *Proceedings of IFIP Network Traffic Measurement and Analysis Conference*, TMA 2019, pages 193–194, June 2019. doi:10.23919/TMA.2019.8784546

[157] Dominik Scholz, Andreas Oeldemann, **Fabien Geyer**, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. Cryptographic Hashing in P4 Data Planes. In *Proceedings of the 2nd P4 Workshop in Europe*, EuroP4 2019, September 2019. doi:10.1109/ANCS.2019.8901886

[160] Charles Shelbourne, Leonardo Linguaglossa, Aldo Lipani, Tianzhu Zhang, and **Fabien Geyer**. On the Learnability of Software Router Performance via CPU Measurements. In *Proceedings of the 2019 CoNEXT Student Workshop*, pages 23–25, December 2019. doi:10.1145/3360468.3366776

2020

[21] Steffen Bondorf and **Fabien Geyer**. Virtual Cross-Flow Detouring in the Deterministic Network Calculus Analysis. In *IFIP Networking 2020*, pages 554–558, June 2020. ISBN 978-3-903176-28-7

[64] **Fabien Geyer** and Steffen Bondorf. On the Robustness of Deep Learning-predicted Contention Models for Network Calculus. In *Proceedings of the 25th IEEE Symposium on Computers and Communications*, ISCC 2020, July 2020. doi:10.1109/ISCC50000.2020.9219693

[98] Cansu Gözde Karadeniz, **Fabien Geyer**, Thomas Multerer, and Dominic Schupke. Precise UWB-Based Localization for Aircraft Sensor Nodes. In *Proceedings of the IEEE/AIAA 39th Digital Avionics Systems Conference*, DASC 2020, October 2020. doi:10.1109/DASC50938.2020.9256793

2021

[11] Aygün Baltaci, Markus Klügel, **Fabien Geyer**, Svetoslav Duhovnikov, Vaibhav Bajpai, Jörg Ott, and Dominic Schupke. Experimental UAV Data Traffic Modeling and Network Performance Analysis. In *Proceedings of the 40th IEEE International Conference on Computer Communications*, INFOCOM 2021, May 2021. doi:10.1109/INFOCOM42981.2021.9488878

[75] **Fabien Geyer**, Alexander Scheffler, and Steffen Bondorf. Tightening Network Calculus Delay Bounds by Predicting Flow Prolongations in the FIFO Analysis. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS 2021, May 2021. doi:10.1109/RTAS52030.2021.00021

[65] **Fabien Geyer** and Steffen Bondorf. Graph-based Deep Learning for Fast and Tight Network Calculus Analyses. *IEEE Transactions on Network Science and Engineering*, 8(1):75–88, January 2021. doi:10.1109/TNSE.2020.3025806

[139] Peter Okelmann, Leonardo Linguaglossa, **Fabien Geyer**, Paul Emmerich, and Georg Carle. Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning. In *Proceedings of the 7th IEEE International Conference on Network Softwarization*, NetSoft, June 2021. doi:10.1109/NetSoft51509.2021.9492668

B. LIST OF RESEARCH ARTIFACTS

Additionally to scientific publications listed in Appendix A, research artifacts listed in Table B.1 have also been published, namely datasets and code. Those were published in order to reproduce the published results, motivate other researchers to improve the published methods, and provide tools for the research community.

Publication	Type	URL
[69]	Dataset	https://github.com/fabgeyer/dataset-itc30nc
[62]	Dataset	https://github.com/fabgeyer/dataset-infocom2019
[70]	Dataset	https://github.com/fabgeyer/dataset-networking2019
[70]	Code	https://github.com/fabgeyer/deepmpls
[63][64][65]	Dataset	https://github.com/fabgeyer/dataset-deeptma-extension
[11]	Code	https://github.com/aygunbaltaci/AVIATOR
[75]	Dataset	https://github.com/fabgeyer/dataset-rtas2021
[139]	Code	https://github.com/pogobanane/vpp-testing

Tab. B.1: Research artifacts published in the scope of the habilitation work

C. GLOSSARY

AFDX Avionics Full DupleX Switched Ethernet
ASIC Application-Specific Integrated Circuit
AVB Audio-Video Bridging
BGP Border Gateway Protocol
BNN binary neural network
CBN Causal Bayesian Networks
CNN convolutional neural network
CPU Central Processing Unit
CRC Cyclic Redundancy Check
DEBORAH Delay Bound Rating AlgoritHm
DNC deterministic network calculus
DPDK Dataplane Development Kit
DSL Domain Specific Language
EIB Explicit Intermediate Bounds
FA Forward End-To-End Delay Approach
FFA Feed-Forward Analysis
FFNN feed-forward neural network
FIFO First In First Out
foi flow of interest
FP Flow Prolongation
FPGA Field Programmable Gate Array
GGNN gated graph neural network
GNN graph neural network
GPU Graphic Processing Unit
GQNN Graph-Query Neural Network
GRU Gated Recurrent Unit
HSA Header Space Analysis
HTTP Hypertext Transfer Protocol
INT In-Band Network Telemetry
LP linear program
LSTM Long Short-Term Memory
LUDB Least Upper Delay Bound
mcastFFA multicast Feed-Forward Analysis
ML machine learning
MPC model predictive control
MPLS Multiprotocol Label Switching
NC network calculus
NIC Network Interface Card

NN	neural network
NPU	Network Processing Unit
OSPF	Open Shortest Path First
PBOO	Pay Burst Only Once
PCC	Performance-oriented Congestion Control
PDA	pushdown automaton
PMOO	Pay Multiplexing Only Once
PMOOA	Pay Multiplexing Only Once Analysis
PSA	Portable Switch Architecture
QoE	Quality-of-Experience
QoS	Quality-of-Service
RAM	Random Access Memory
RIP	Routing Information Protocol
RNN	Recurrent Neural Network
RTT	round-trip time
SAT	Boolean satisfiability problem
SDN	Software Defined Networking
SFA	Separate Flow Analysis
SHA	Secure Hash Algorithm
SNC	stochastic network calculus
SVM	Support Vector Machine
SVR	Support Vector Regression
T4P4S	Translator for P4 Switches
TA	Trajectory Approach
TCP	Transmission Control Protocol
TE	Traffic Engineering
TFA	Total Flow Analysis
TMA	Tandem Matching Analysis
TSN	Time-Sensitive Networking
UDP	User Datagram Protocol
ULP	Unique Linear Program
VHDL	VHSIC Hardware Description Language
VNF	virtual network function
VPP	Vector Packet Processing

BIBLIOGRAPHY

[1] Muhammad Adnan, Jean-Luc Scharbarg, and Christian Fraboul. Minimizing the search space for computing exact worst-case delays of AFDX periodic flows. In *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems*, SIES, pages 294–301, June 2011. doi:10.1109/SIES.2011.5953673.

[2] Aeronautical Radio Inc. ARINC Specification 664P7-1: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network, September 2009.

[3] Rajeev Agrawal, Rene L. Cruz, Clayton Okino, and Rajendran Rajan. Performance Bounds for Flow Control Protocols. *IEEE/ACM Transactions on Networking*, 7(3):310–323, June 1999. ISSN 1063-6692. doi:10.1109/90.779197.

[4] Hussein Al-Zubaidy, Jörg Liebeherr, and Almut Burchard. A (min, ×) network calculus for multi-hop fading channels. In *Proc. of IEEE INFOCOM*, pages 1833–1841, April 2013. doi:10.1109/INFCOM.2013.6566982.

[5] Moussa Amrani, Levi Lúcio, and Adrien Bibal. ML + FV = ♡? A Survey on the Application of Machine Learning to Formal Verification. June 2018. arXiv:1806.03600.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, January 2014. doi:10.1145/2578855.2535862.

[7] Davide Andreoletti, Sebastian Troia, Francesco Musumeci, Silvia Giordano, Guido Maier, and Massimo Tornatore. Network traffic prediction based on diffusion convolutional recurrent neural networks. In *Proceedings of the 38th IEEE Conference on Computer Communications Workshops*, INFOCOM Workshops, April 2019. doi:10.1109/INFCOMW.2019.8845132.

[8] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *Lecture Notes in Computer Science*, pages 489–508. Springer Berlin Heidelberg, December 2012. doi:10.1007/978-3-642-34931-7_28.

[9] François Baccelli and Dohy Hong. TCP is Max-Plus Linear and what it tells us on its throughput. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 219–230. ACM, October 2000. doi:10.1145/347057.347548.

[10] Mahmoud Bahnasy, Fenglin Li, Shihan Xiao, and Xiangle Cheng. DeepBGP: A machine learning approach for BGP configuration synthesis. In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI ’20, August 2020. doi:10.1145/3405671.3405816.

[11] Aygün Baltaci, Markus Klügel, Fabien Geyer, Svetoslav Duhovnikov, Vaibhav Bajpai, Jörg Ott, and Dominic Schupke. Experimental UAV Data Traffic Modeling and Network Performance Analysis. In *Proceedings of the 40th IEEE International Conference on Computer Communications*, INFOCOM 2021, May 2021. doi:10.1109/INFOCOM42981.2021.9488878.

[12] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. June 2018. arXiv:1806.01261.

[13] Henri Bauer, Jean-Luc Scharbarg, and Christian Fraboul. Applying and optimizing trajectory approach for performance evaluation of AFDX avionics network. In *Proceedings of the IEEE Conference on Emerging Technologies Factory Automation*, ETFA 2009, pages 1–8, September 2009. doi:10.1109/ETFA.2009.5347083.

[14] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *Proceedings of the 26th IEEE International Conference on Network Protocols*, ICNP, September 2018. doi:10.1109/icnp.2018.00047.

[15] Luca Bisti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Estimating the worst-case delay in FIFO tandems using network calculus. In *Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools, October 2008. doi:10.4108/ICST.VALUETOOLS2008.4388.

[16] Luca Bisti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. DEBORAH: A tool for worst-case analysis of FIFO tandems. In *Proceedings of the 2010 International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, ISoLA, pages 152–168, October 2010. doi:10.1007/978-3-642-16558-0_15.

[17] Luca Bisti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Numerical analysis of worst-case end-to-end delay bounds in FIFO tandem networks. *Real-Time Systems*, 48(5): 527–569, 2012. doi:10.1007/s11241-012-9153-1.

[18] Steffen Bondorf. Better bounds by worse assumptions - improving network calculus accuracy by adding pessimism to the network model. In *Procdeedings of the 2017 IEEE International Conference on Communications*, ICC, May 2017. doi:10.1109/icc.2017.7996996.

[19] Steffen Bondorf and Fabien Geyer. Generalizing Network Calculus Analysis to Derive Performance Guarantees for Multicast Flows. In *Proceedings of the 10th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2016, October 2016. doi:10.4108/eai.25-10-2016.2266598.

[20] Steffen Bondorf and Fabien Geyer. *Deterministic Network Calculus Analysis of Multicast Flows*, pages 63–78. Springer International Publishing, Cham, 2019. ISBN 978-3-319-92378-9. doi:10.1007/978-3-319-92378-9_5.

[21] Steffen Bondorf and Fabien Geyer. Virtual Cross-Flow Detouring in the Deterministic Network Calculus Analysis. In *IFIP Networking 2020*, pages 554–558, June 2020. ISBN 978-3-903176-28-7.

[22] Steffen Bondorf and Jens B. Schmitt. The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2014)*, December 2014. doi:10.4108/icst.valuetools.2014.258167.

[23] Steffen Bondorf and Jens B. Schmitt. Should network calculus relocate? an assessment of current algebraic and optimization-based analyses. In *Proceedings of Quantitative Evaluation of Systems*, QEST, pages 207–223, September 2016. doi:10.1007/978-3-319-43425-4_15.

[24] Steffen Bondorf, Paul Nikolaus, and Jens B. Schmitt. Quality and cost of deterministic network calculus – design and evaluation of an accurate and fast analysis. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 1(1):34, June 2017. doi:10.1145/3084453.

[25] Steffen Bondorf, Paul Nikolaus, and Jens B. Schmitt. Catching Corner Cases in Network Calculus – Flow Segregation Can Improve Accuracy. In *Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems*, February 2018. doi:10.1007/978-3-319-74947-1_15.

[26] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110, August 2013. doi:10.1145/2486001.2486011.

[27] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014. ISSN 0146-4833. doi:10.1145/2656877.2656890.

[28] Anne Bouillard, Bertrand Cottenceau, Bruno Gaujal, Laurent Hardouin, Sébastien Lagrange, and Mehdi Lhommeau. COINC Library: a toolbox for the Network Calculus. In *Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS*, number 35. ICST, October 2009. doi:10.4108/ICST.VALUETOOLS2009.8045.

[29] Anne Bouillard, Laurent Jouhet, and Eric Thierry. Tight Performance Bounds in the Worst-Case Analysis of Feed-Forward Networks. In *Proceedings of the 29th IEEE International Conference on Computer Communications*, INFOCOM 2010, March 2010. doi:10.1109/INFCOM.2010.5461912.

[30] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons, Inc., October 2018. doi:10.1002/9781119440284.

[31] Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 671–678. Morgan-Kaufmann, 1994.

[32] Marc Boyer. NC-maude: a rewriting tool to play with network calculus. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin Heidelberg, October 2010. ISBN 978-3-642-16557-3. doi:10.1007/978-3-642-16558-0_14.

[33] Marc Boyer and Christian Fraboul. Tightening end to end delay upper bound for AFDX network calculus with rate latency FIFO servers using network calculus. In *Proceedings of the 2008 IEEE International Workshop on Factory Communication Systems*, WFCS, pages 11–20, May 2008. doi:10.1109/WFCS.2008.4638728.

[34] Marc Boyer and Pierre Roux. Embedding network calculus and event stream theory in a common model. In *Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA, pages 1–8, September 2016. doi:10.1109/ETFA.2016.7733565.

[35] Marc Boyer, Jörn Migge, and Marc Fumey. PEGASE - A Robust and Efficient Tool for Worst-Case Network Traversal Time Evaluation on AFDX. In *Proceedings of SAE Aerotech 2011*, January 2011. doi:10.4271/2011-01-2711.

[36] Gordon Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro*, 34(1):8–18, January 2014.

[37] Broadcom. NPL – Network Programming Language Specification v1.3. Version 1.3, June 2019. URL `https://nplang.org/npl/specifications/`.

[38] Mihai Budiu and Chris Dodd. The $p4_{16}$ programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, September 2017. doi:10.1145/3139645.3139648.

[39] Cheng-Shang Chang. On Deterministic Traffic Regulation and Service Guarantees: A Systematic Approach by Filtering. *IEEE Transactions on Information Theory*, 44(3):1097–1110, May 1998. doi:10.1109/18.669173.

[40] Cheng-Shang Chang. *Performance Guarantees in Communication Networks*. Springer, 2000. ISBN 978-1-4471-0459-9. doi:10.1007/978-1-4471-0459-9.

[41] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th ACM International Conference on Emerging Networking Experiments And Technologies*, CoNEXT, December 2019. doi:10.1145/3359989.3365408.

[42] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, October 2014. doi:10.3115/v1/D14-1179.

[43] Rene L. Cruz. A Calculus for Network Delay, Part I. Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991. doi:10.1109/18.61109.

[44] Rene L. Cruz. A Calculus for Network Delay, Part II. Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991. doi:10.1109/18.61110.

[45] Hugo Daigmorte, Marc Boyer, and Luxi Zhao. Modelling in network calculus a TSN architecture mixing Time-Triggered, Credit Based Shaper and Best-Effort queues. working paper or preprint, June 2018. URL https://hal.archives-ouvertes.fr/hal-01814211.

[46] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2):18–24, April 2016. doi:10.1145/2935634.2935638.

[47] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, August 2020. doi:10.1109/tnet.2020.2992106.

[48] Joan Adrià Ruiz De Azua and Marc Boyer. Complete Modelling of AVB in Network Calculus Framework. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 55:55–55:64, October 2014. ISBN 978-1-4503-2727-5. doi:10.1145/2659787.2659810.

[49] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 12, pages 141–154, April 2012.

[50] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'15, pages 395–408, May 2015.

[51] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P. Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 343–356. USENIX Association, April 2018.

[52] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the Internet Measurement Conference*, IMC'15, October 2015.

[53] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems. In *Proceedings of the 14th International Conference on Networks*, ICN 2015, pages 78–83, April 2015.

[54] Nick Feamster and Jennifer Rexford. Why (and How) Networks Should Run Themselves. October 2017. arXiv:1710.11583v1.

[55] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. March 2019. arXiv:1903.02428.

[56] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks. February 2016. arXiv:1602.02672v1.

[57] Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize the AFDX network. In *ERTS 2006 - 3rd European Congress ERTS Embedded real-time software*, January 2006.

[58] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. ACM, November 2020. doi:10.1145/3422604.3425928.

[59] Fabien Geyer. Routing Optimization for SDN Networks Based on Pivoting Rules for the Simplex Algorithm. In *Proceedings of the 13th International Conference on Design of Reliable Communication Networks*, DRCN 2017, pages 47–54, March 2017. ISBN 978-3-8007-4383-4.

[60] Fabien Geyer. Performance Evaluation of Network Topologies using Graph-Based Deep Learning. In *Proceedings of the 11th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2017, pages 20–27, December 2017. doi:10.1145/3150928.3150941.

[61] Fabien Geyer. DeepComNet: Performance Evaluation of Network Topologies using Graph-Based Deep Learning. *Performance Evaluation*, 130:1–16, April 2019. ISSN 0166-5316. doi:10.1016/j.peva.2018.12.003.

[62] Fabien Geyer and Steffen Bondorf. DeepTMA: Predicting Effective Contention Models for Network Calculus using Graph Neural Networks. In *Proceedings of the 38th IEEE International Conference on Computer Communications*, INFOCOM 2019, April 2019. doi:10.1109/INFOCOM.2019.8737496.

[63] Fabien Geyer and Steffen Bondorf. On the Robustness of Deep Learning-predicted Contention Models for Network Calculus. *arxiv:1911.10522*, November 2019.

[64] Fabien Geyer and Steffen Bondorf. On the Robustness of Deep Learning-predicted Contention Models for Network Calculus. In *Proceedings of the 25th IEEE Symposium on Computers and Communications*, ISCC 2020, July 2020. doi:10.1109/ISCC50000.2020.9219693.

[65] Fabien Geyer and Steffen Bondorf. Graph-based Deep Learning for Fast and Tight Network Calculus Analyses. *IEEE Transactions on Network Science and Engineering*, 8(1):75–88, January 2021. doi:10.1109/TNSE.2020.3025806.

[66] Fabien Geyer and Georg Carle. Network Engineering for Real-Time Networks: Comparison of Automotive and Aeronautic Industries Approaches. *IEEE Communications Magazine*, 54(2):106–112, February 2016. ISSN 0163-6804. doi:10.1109/MCOM.2016.7402269.

[67] Fabien Geyer and Georg Carle. Towards Automatic Performance Optimization of Networks Using Machine Learning. In *Proceedings of the 17th International Network Strategy and Planning Symposium*, NETWORKS 2016, September 2016. doi:10.1109/NETWKS.2016.7751147.

[68] Fabien Geyer and Georg Carle. Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning. In *Proceedings of the 2018 SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, Big-DAMA 2018, pages 40–45, Budapest, Hungary, August 2018. doi:10.1145/3229607.3229610.

[69] Fabien Geyer and Georg Carle. The Case for a Network Calculus Heuristic: Using Insights from Data for Tighter Bounds. In *Proceedings of the 2018 International Workshop on Network Calculus and Applications*, NetCal 2018, pages 43–48, Vienna, Austria, September 2018. doi:10.1109/ITC30.2018.10060.

[70] Fabien Geyer and Stefan Schmid. DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning. In *IFIP Networking 2019*, May 2019. doi:10.23919/IFIPNetworking.2019.8816842.

[71] Fabien Geyer and Max Winkel. Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications. In *Proceedings of the 9th European Congress Embedded Real Time Software and Systems*, ERTS 2018, February 2018.

[72] Fabien Geyer, Alexandros Elefsiniotis, Dominic Schupke, and Stefan Schneele. Multi-Objective Optimization of Aircraft Networks for Weight, Performance and Survivability. In *Proceedings of the 7th International Workshop on Reliable Networks Design and Modeling*, RNDM 2015, pages 345–350, October 2015. ISBN 978-1-4673-8051-5. doi:10.1109/RNDM.2015.7325251.

[73] Fabien Geyer, Max Winkel, and Stefan Schneele. Rapid Prototyping of Packet Processing Devices for Aeronautical Applications. In Frank Thielecke Otto von Estorff, editor, *Proceedings of the 6th International Workshop on Aircraft System Technologies*, AST 2017, pages 273–282. Shaker-Verlag, February 2017. ISBN 978-3-8440-5086-8.

[74] Fabien Geyer, Holger Kinkelin, Hendrik Leppelsack, Stefan Liebald, Dominik Scholz, Georg Carle, and Dominic Schupke. Performance Perspective on Private Distributed Ledger Technologies for Industrial Networks. In *Proceedings of the International Conference on Networked Systems*, NetSys 2019, March 2019. doi:10.1109/NetSys.2019.8854512.

[75] Fabien Geyer, Alexander Scheffler, and Steffen Bondorf. Tightening Network Calculus Delay Bounds by Predicting Flow Prolongations in the FIFO Analysis. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS 2021, May 2021. doi:10.1109/RTAS52030.2021.00021.

[76] Ran Giladi. *Network Processors: Architecture, Programming, and Implementation*. Morgan Kaufmann, 2008.

[77] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A New Model for Learning in Graph Domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, volume 2 of *IJCNN'05*, pages 729–734, August 2005. doi:10.1109/IJCNN.2005.1555942.

[78] Daniele Grattarola and Cesare Alippi. Graph neural networks in TensorFlow and keras with spektral. *IEEE Computational Intelligence Magazine*, 16(1):99–106, February 2021. doi:10.1109/mci.2020.3039072.

[79] Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, Institut National Polytechnique de Toulouse, September 2004.

[80] Nan Guan and Wang Yi. Finitary real-time calculus: Efficient performance analysis of distributed embedded systems. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, RTSS, pages 330–339, December 2013. doi:10.1109/RTSS.2013.40.

[81] Peter Heise, Fabien Geyer, and Roman Obermaisser. Deterministic OpenFlow: Performance Evaluation of SDN Hardware for Avionic Networks. In *Proceedings of the 11th International Conference on Network and Service Management*, CNSM 2015, pages 372–377, November 2015. doi:10.1109/CNSM.2015.7367385.

[82] Peter Heise, Fabien Geyer, and Roman Obermaisser. TSimNet: An industrial Time Sensitive Networking simulation framework based on OMNeT++. In *Proceedings of the 8th IFIP International Conference on New Technologies, Mobility and Security*, NTMS, November 2016. doi:10.1109/NTMS.2016.7792488.

[83] Peter Heise, Marc Lasch, Fabien Geyer, and Roman Obermaisser. Self-Configuring Real-Time Communication Network based on OpenFlow. In *Proceedings of the 22nd IEEE International Symposium on Local and Metropolitan Area Networks*, LANMAN 2016, June 2016. doi:10.1109/LANMAN.2016.7548851.

[84] Peter Heise, Fabien Geyer, and Roman Obermaisser. Self-Configuring Deterministic Network with In-Band Configuration Channel. In *Proceedings of the 4th IEEE International Conference on Software Defined Systems*, SDS 2017, May 2017. doi:10.1109/SDS.2017.7939158.

[85] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. doi:10.1162/neco.1997.9.8.1735.

[86] Hadrien Hours, Ernst W. Biersack, and Patrick Loiseau. A Causal Approach to the Study of TCP Performance. *ACM Transactions on Intelligent Systems and Technology*, 7(2):25:1–25:25, January 2016. doi:10.1145/2770878.

[87] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, August 2018. doi:10.1145/3230543.3230559.

[88] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4→NetFPGA workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2019. doi:10.1145/3289602.3293924.

[89] Netronome Systems Inc. NFP-4000 Theory of Operation. Technical Report WP-NFP-4000-TOO-1/2016, 2016.

[90] Intel. Intel Deep Insight Network Analytics Software. URL `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/network-analytics/deep-insight.html`.

[91] Van Jacobson. Modified TCP congestion avoidance algorithm. end2end-interest mailing list, April 1990.

[92] Benedikt Jaeger, Dominik Scholz, Daniel Raumer, Fabien Geyer, and Georg Carle. Reproducible Measurements of TCP BBR Congestion Control. *Computer Communications*, May 2019. doi:10.1016/j.comcom.2019.05.011.

[93] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. October 2018. arXiv:1810.03259.

[94] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3050–3059, June 2019.

[95] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT, December 2018. doi:10.1145/3281411.3281432.

[96] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, October 2017. doi:10.1145/3132747.3132764.

[97] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 35–49. USENIX Association, April 2018. doi:10.5555/3307441.3307445.

[98] Cansu Gözde Karadeniz, Fabien Geyer, Thomas Multerer, and Dominic Schupke. Precise UWB-Based Localization for Aircraft Sensor Nodes. In *Proceedings of the IEEE/AIAA 39th Digital Avionics Systems Conference*, DASC 2020, October 2020. doi:10.1109/DASC50938.2020.9256793.

[99] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR, March 2016. doi:10.1145/2890955.2890968.

[100] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 113–126, April 2012. ISBN 978-931971-92-8. doi:10.5555/2228298.2228311.

[101] Georges Kemayo, Frédéric Ridouard, Henri Bauer, and Pascal Richard. A Forward end-to-end delays Analysis for packet switched networks. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 65:65–65:74, October 2014. ISBN 978-1-4503-2727-5. doi:10.1145/2659787.2659801.

[102] Georges Kemayo, Nassima Benammar, Frédéric Ridouard, Henri Bauer, and Pascal Richard. Improving AFDX End-to-End Delays Analysis. In *Proceedings of the 20th IEEE*

Conference on Emerging Technologies Factory Automation, ETFA, pages 1–8, September 2015. doi:10.1109/ETFA.2015.7301463.

[103] Sven Kerschbaum, Kai-Steffen Hielscher, and Reinhard German. The need for shaping non-time-critical data in PROFINET networks. In *Proceedings of the 14th IEEE International Conference on Industrial Informatics*, INDIN, July 2016. doi:10.1109/INDIN.2016.7819151.

[104] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, October 2011. ISSN 0733-8716. doi:10.1109/JSAC.2011.111002.

[105] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[106] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *ACM SIGCOMM Computer Communication Review*, SIGCOMM '16, pages 629–630, August 2016. ISBN 978-1-4503-4193-6. doi:10.1145/2934872.2959080.

[107] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *Proceedings of the 7th ACM International Conference on Embedded Software*, EMSOFT, pages 107–116, October 2009. doi:10.1145/1629335.1629351.

[108] Kai Lampka, Steffen Bondorf, and Jens B. Schmitt. Achieving efficiency without sacrificing model accuracy: Network calculus on compact domains. In *Proceedings of the 24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 313–318, September 2016. doi:10.1109/MASCOTS.2016.9.

[109] Kai Lampka, Steffen Bondorf, Jens B. Schmitt, Nan Guan, and Wang Yi. Generalized finitary real-time calculus. In *Proceedings of the 36th IEEE International Conference on Computer Communications*, INFOCOM 2017, pages 1–9, May 2017. doi:10.1109/INFOCOM.2017.8056981.

[110] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2): 133–169, May 1998. doi:10.1145/279227.279229.

[111] Jonatan Langlet. Towards machine learning inference in the data plane, June 2019.

[112] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6. ACM, October 2010. doi:10.1145/1868447.1868466.

[113] Jean-Yves Le Boudec. A theory of traffic regulators for deterministic networks with application to interleaved regulators. *IEEE/ACM Transactions on Networking*, 26(6):2721–2733, December 2018. doi:10.1109/tnet.2018.2875191.

[114] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001. ISBN 978-3-540-42184-9. doi:10.1007/3-540-45318-0.

[115] Yann Le Cun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, December 1989. doi:10.1162/neco.1989.1.4.541.

[116] Yann Le Cun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

[117] Mengyuan Lee, Guanding Yu, and Geoffrey Ye Li. Graph embedding based wireless link scheduling with few training samples. June 2019. arXiv:1906.02871.

[118] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th ACM International Symposium on Computer Architecture*, June 2019. doi:10.1145/3307650.3322259.

[119] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. In *Proceedings of the 4th International Conference on Learning Representations*, ICLR'2016, April 2016.

[120] Jörg Liebeherr. Duality of the max-plus and min-plus network calculus. *Foundations and Trends® in Networking*, 11(3-4):139–282, 2017. doi:10.1561/1300000059.

[121] Leonardo Linguaglossa, Fabien Geyer, Wenqin Shao, Frank Brockners, and Georg Carle. Demonstrating the Cost of Collecting In-Network Measurements for High-Speed VNFs. In *Proceedings of IFIP Network Traffic Measurement and Analysis Conference*, TMA 2019, pages 193–194, June 2019. doi:10.23919/TMA.2019.8784546.

[122] Tieu Long Mai and Nicolas Navet. Deep learning to predict the feasibility of priority-based ethernet network configurations. Technical report, University of Luxembourg, August 2020.

[123] Tieu Long Mai and Nicolas Navet. Improvements to deep-learning-based feasibility prediction of switched ethernet network configurations. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, RTNS, April 2021.

[124] Lisa Maile, Kai-Steffen Hielscher, and Reinhard German. Network calculus results for TSN: An introduction. In *Proceedings of the Information Communication Technologies Conference*, May 2020. doi:10.1109/ictc49638.2020.9123308.

[125] Bomin Mao, Zubair Md. Fadlullah, Fengxiao Tang, Nei Kato, Osamu Akashi, Takeru Inoue, and Kimihiro Mizutani. Routing or Computing? The Paradigm Shift Towards Intelligent Computer Network Packet Transmission Based on Deep Learning. *IEEE Transactions on Computers*, 66(11):1946–1960, November 2017. doi:10.1109/TC.2017.2709742.

[126] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 197–210, August 2017. doi:10.1145/3098822.3098843.

[127] Steven Martin and Pascale Minet. Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006. doi:10.1109/IPDPS.2006.1639424.

[128] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960. ISSN 0367-9950. doi:10.1109/TEC.1960.5221603.

[129] Mao Miao, Wenxue Cheng, Fengyuan Ren, and Jing Xie. Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications*, December 2016. doi:10.1109/hpcc-smartcity-dss.2016.0106.

[130] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, August 2017. doi:10.1145/3098822.3098824.

[131] Ahlem Mifdaoui and Hamdi Ayed. WOPANets: A tool for WOrst case performance analysis of embedded networks. In *Proceedings of the 15th IEEE International Workshop on Computer Aided Modeling, Analysis and Design of Communication Links and Networks*, CAMAD, December 2010. doi:10.1109/CAMAD.2010.5686958.

[132] Jörn Migge. *L'ordonnancement sous contraintes temps-réel un modèle à base de trajectoires*. PhD thesis, INRIA Sophia Antipolis, November 1999.

[133] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. A Machine Learning Approach to TCP Throughput Prediction. *IEEE/ACM Transactions on Networking*, 18(4):1026–1039, August 2010. ISSN 1063-6692. doi:10.1109/TNET.2009.2037812.

[134] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 151–160. ACM, October 2000. doi:10.1145/347057.347421.

[135] Ehsan Mohammadpour, Eleni Stai, and Jean-Yves Le Boudec. Improved delay bound for a service curve element with known transmission rate. *IEEE Networking Letters*, 1(4):156–159, December 2019. doi:10.1109/lnet.2019.2927143.

[136] Dritan Nace and Michał Pióro. Max-Min Fairness and Its Applications to Routing and Load-Balancing in Communication Networks: A Tutorial. *IEEE Communications Surveys and Tutorials*, 10(4):5–17, 2008. ISSN 1553-877X. doi:10.1109/SURV.2008.080403.

[137] Kota Nakashima, Shotaro Kamiya, Kazuki Ohtsu, Koji Yamamoto, Takayuki Nishio, and Masahiro Morikura. Deep reinforcement learning-based channel allocation for wireless lans with graph convolutional networks. May 2019. arXiv:1905.07144.

[138] Paul Nikolaus and Jens Schmitt. Improving delay bounds in the stochastic network calculus by using less stochastic inequalities. In *Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools*, ValueTools, pages 96–103, May 2020. doi:10.1145/3388831.3388848.

[139] Peter Okelmann, Leonardo Linguaglossa, Fabien Geyer, Paul Emmerich, and Georg Carle. Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning. In *Proceedings of the 7th IEEE International Conference on Network Softwarization*, NetSoft, June 2021. doi:10.1109/NetSoft51509.2021.9492668.

[140] Jitendra Padhye, Victor Firoiu, Don F. Towsley, and James F. Kurose. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, April 2000. ISSN 1063-6692. doi:10.1109/90.842137.

[141] Leonid Peshkin and Virginia Savova. Reinforcement Learning for Adaptive Routing. In *Proceedings of the 2002 International Joint Conference on Neural Networks*, IJCNN, pages 1825–1830, May 2002. doi:10.1109/IJCNN.2002.1007796.

[142] Fast Data Project. What is VPP?, Accessed 2022/06/08. URL https://wiki.fd.io/view/VPP.

[143] Rene Queck. Analysis of Ethernet AVB for Automotive Networks using Network Calculus. In *Proceedings of the 2012 IEEE International Conference on Vehicular Electronics and Safety*, ICVES, pages 61–67. IEEE, July 2012. doi:10.1109/ICVES.2012.6294261.

[144] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. doi:10.1038/323533a0.

[145] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, October 2020. doi:10.1109/JSAC.2020.3000405.

[146] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the AI accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, August 2018. doi:10.1145/3229591.3229594.

[147] Kaku Sawada, Daisuke Kotani, and Yasuo Okabe. Network routing optimization based on machine learning using graph networks robust against topology change. In *Proceedings of the 2020 International Conference on Information Networking*, ICOIN, January 2020. doi:10.1109/ICOIN48656.2020.9016573.

[148] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, January 2009. doi:10.1109/TNN.2008.2005605.

[149] Alexander Scheffler, Markus Fögen, and Steffen Bondorf. The deterministic network calculus analysis: Reliability insights and performance improvements. In *Proceedings of the 23rd IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, CAMAD, pages 1–6, September 2018. doi:10.1109/CAMAD.2018.8514938.

[150] Henrik Schioler, Hans P. Schwefel, and Martin B. Hansen. CyNC – A MATLAB/SimuLink Toolbox for Network Calculus. In *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools ’07, pages 60:1–60:10. ICST, October 2007. doi:10.4108/valuetools.2007.1804.

[151] Stefan Schmid and Jiří Srba. Polynomial-time what-if analysis for prefix-manipulating MPLS networks. In *Proc. IEEE INFOCOM*, April 2018. doi:10.1109/INFOCOM.2018.8486261.

[152] Jens B. Schmitt and Frank A. Zdarsky. The DISCO Network Calculator - A Toolbox for Worst Case Analysis. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools (Valuetools'06)*. ICST, ACM, October 2006. doi:10.1145/1190095.1190105.

[153] Jens B. Schmitt, Frank A. Zdarsky, and Markus Fidler. Delay Bounds under Arbitrary Multiplexing: When Network Calculus Leaves You in the Lurch.... In *Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM 2008, pages 1669–1677, April 2008. doi:10.1109/INFOCOM.2008.228.

[154] Jens B. Schmitt, Frank A. Zdarsky, and Ivan Martinovic. Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once. In *Proceedings of the 14th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems*, MMB 2008, pages 1–15, March 2008.

[155] Dominik Scholz, Fabien Geyer, Sebastian Gallenmüller, and Georg Carle. Rapid Prototyping of Avionic Applications Using P4. In *5th P4 Workshop*, Stanford, CA, USA, June 2018.

[156] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking 2018*, May 2018. doi:10.23919/IFIPNetworking.2018.8696830.

[157] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. Cryptographic Hashing in P4 Data Planes. In *Proceedings of the 2nd P4 Workshop in Europe*, EuroP4 2019, September 2019. doi:10.1109/ANCS.2019.8901886.

[158] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. February 2018. arXiv:1802.03685v1.

[159] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 1–16. USENIX Association, April 2018. doi:10.5555/3307441.3307443.

[160] Charles Shelbourne, Leonardo Linguaglossa, Aldo Lipani, Tianzhu Zhang, and Fabien Geyer. On the Learnability of Software Router Performance via CPU Measurements. In *Proceedings of the 2019 CoNEXT Student Workshop*, pages 23–25, December 2019. doi:10.1145/3360468.3366776.

[161] Yifei Shen, Yuanming Shi, Jun Zhang, and Khaled B. Letaief. A graph neural network approach for scalable wireless power control. July 2019. arXiv:1907.08487.

[162] Giuseppe Siracusano and Roberto Bifulco. In-network neural networks. January 2018. arXiv:1801.05731.

[163] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running neural networks on the NIC. September 2020. arXiv:2009.02353.

[164] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 44–57, August 2016. ISBN 978-1-4503-4193-6. doi:10.1145/2934872.2934899.

[165] Haoyu Song. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in Software Defined Networking*, pages 127–132, August 2013. doi:10.1145/2491185.2491190.

[166] Dimitrios Stiliadis and Anujan Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, October 1998. doi:10.1109/90.731196.

[167] Sainbayar Sukhbaatar, Rob Fergus, and others. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252, 2016.

[168] Taisei Suzuki, Yuichi Yasuda, Ryo Nakamura, and Hiroyuki Ohsaki. On estimating communication delays using graph convolutional networks with semi-supervised learning. In *2020 International Conference on Information Networking (ICOIN)*, January 2020. doi:10.1109/icoin48656.2020.9016603.

[169] Mukarram Bin Tariq, Kaushik Bhandankar, Vytautas Valancius, Amgad Zeitoun, Nick Feamster, and Mostafa Ammar. Answering "What-If" Deployment and Configuration Questions With WISE: Techniques and Deployment Experience. *IEEE/ACM Transactions on Networking*, 21(1):1–13, February 2013. ISSN 1558-2566. doi:10.1109/TNET.2012.2230448.

[170] The P4 Language Consortium. The P4 Language Specification. Version 1.0.5, November 2018. URL `https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf`.

[171] The P4 Language Consortium. $P4_{16}$ Language Specification. Version 1.2.1, June 2020. URL `https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf`.

[172] The P4.org Applications Working Group. In-band Network Telemetry (INT) Dataplane Specification. Version 2.1, October 2020. URL `https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf`.

[173] The P4.org Applications Working Group. Telemetry Report Format Specification. Version 2.0, October 2020. URL `https://raw.githubusercontent.com/p4lang/p4-applications/master/docs/telemetry_report_v2_0.pdf`.

[174] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, May 2000. doi:10.1109/ISCAS.2000.858698.

[175] Guibin Tian and Yong Liu. Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 109–120, December 2012. ISBN 978-1-4503-1775-7. doi:10.1145/2413176.2413190.

[176] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994. ISSN 0165-6074. doi:10.1016/0165-6074(94)90080-9.

[177] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. LaKe: The power of in-network computing. In *Proceedings of the 2018 International Conference on ReConFigurable Computing and FPGAs*, ReConFig, December 2018. doi:10.1109/reconfig.2018.8641696.

[178] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *Proceedings of the 14th EuroSys Conference 2019*, EuroSys '19, March 2019. doi:10.1145/3302424.3303979.

[179] Muhammad Usama, Junaid Qadir, Aunn Raza, Hunain Arif, Kok-lim Alvin Yau, Yehia Elkhatib, Amir Hussain, and Ala Al-Fuqaha. Unsupervised machine learning for networking: Techniques, applications and research challenges. *IEEE Access*, 7:65579–65615, 2019. doi:10.1109/ACCESS.2019.2916648.

[180] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to Route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 185–191, New York, NY, USA, November 2017. ISBN 978-1-4503-5569-8. doi:10.1145/3152434.3152441.

[181] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 6000–6010. Curran Associates, Inc., December 2017.

[182] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *International Conference on Learning Representations*, April 2018.

[183] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4P4S: A target-independent compiler for protocol-independent packet processors. In *Proceedings of the 19th IEEE International Conference on High Performance Switching and Routing*, HPSR, June 2018. doi:10.1109/hpsr.2018.8850752.

[184] Fujun Wang, Zining Cao, Lixing Tan, and Hui Zong. Survey on learning-based formal methods: Taxonomy, applications and possible future directions. *IEEE Access*, 8:108561–108578, 2020. doi:10.1109/ACCESS.2020.3000907.

[185] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. September 2019. arXiv:1909.01315.

[186] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. September 2017. doi:10.1109/MNET.2017.1700200. arXiv:1709.08339v2.

[187] Wei Wang, Yiyang Shao, and Kai Zheng. Muses: Enabling lightweight and diversity for learning based congestion control. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, August 2018. doi:10.1145/3234200.3234202.

[188] Paul J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, October 1990. doi:10.1109/5.58337.

[189] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, August 2013. ISSN 0146-4833. doi:10.1145/2534169.2486020.

[190] David Wolpert, Kagan Tumer, and Jeremy Frank. Using collective intelligence to route internet traffic. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 952–960. MIT Press, 1999.

[191] Shihan Xiao, Haiyan Mao, Bo Wu, Wenjie Liu, and Fenglin Li. Neural packet routing. In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, August 2020. doi:10.1145/3405671.3405813.

[192] Junfeng Xie, F. Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, Chenmeng Wang, and Yunjie Liu. A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges. *IEEE Communications Surveys and Tutorials*, 21(1): 393–430, 2019. doi:10.1109/COMST.2018.2866942.

[193] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets, November 2019. doi:10.1145/3365609.3365864.

[194] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.

[195] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, August 2018. doi:10.1145/3230543.3230544.

[196] Alireza Zamani, Fabien Geyer, Alexandros Elefsiniotis, and Anke Schmeink. Aircraft Network Deployment Optimization with k-Survivability. In *Proceedings of the 11th IEEE International Conference on Advanced Networks and Telecommunications Systems*, ANTS 2017. IEEE, December 2017. doi:10.1109/ANTS.2017.8384192.

[197] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys and Tutorials*, 21(3):2224–2287, 2019. doi:10.1109/COMST.2019.2904897.

[198] Luxi Zhao, Paul Pop, and Silviu S. Craciunas. Worst-case latency analysis for IEEE 802.1Qbv Time Sensitive Networks using network calculus. *IEEE Access*, 6:41803–41815, 2018. doi:10.1109/ACCESS.2018.2858767.

[199] Luxi Zhao, Paul Pop, Zhong Zheng, and Qiao Li. Timing analysis of AVB traffic in TSN networks using network calculus. In *Proceedings of the 2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS. IEEE, April 2018. doi:10.1109/RTAS.2018.00009.

[200] Boyang Zhou, Isaac Howenstine, Siraphob Limprapaipong, and Liang Cheng. A survey on network calculus tools for network infrastructure in real-time systems. *IEEE Access*, 8:223588–223605, 2020. doi:10.1109/ACCESS.2020.3043600.

[201] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014. doi:10.1109/mm.2014.61.

ISBN 978-3-937201-74-0
DOI 10.2313/NET-2022-04-1

ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)