Principled Approaches for Latency Reduction in Networking Systems

by

Benoit Pit-Claudel

Dipl. Ing. École Polytechnique, 2018
M.Sc., Stanford, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR IN PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by:	Benoit Pit-Claudel Department of Electrical Engineering and Computer Science February 28, 2025
Certified by:	Manya Ghobadi Associate Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Certified by:	Muriel Médard NEC Professor of Software Science and Engineering, Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

THESIS COMMITTEE

Thesis Supervisors

Manya Ghobadi
Associate Professor
Electrical Engineering and Computer Science
Muriel Médard
NEC Professor of Software Science and Engineering
Electrical Engineering and Computer Science

Thesis Readers

Mohammad Alizadeh
Associate Professor
Electrical Engineering and Computer Science

Principled Approaches for Latency Reduction in Networking Systems

by

Benoit Pit-Claudel

Submitted to the Department of Electrical Engineering and Computer Science
on February 28, 2025 in partial fulfillment of the requirements for the degree of

DOCTOR IN PHILOSOPHY

ABSTRACT

Modern networks face unprecedented challenges due to exponential growth in traffic demands, driven by AI workloads in datacenters and the ubiquitous adoption of cloud services across the internet. This dissertation addresses three critical challenges in network systems: efficient scheduling of inference tasks, performance optimization in hybrid networks, and memory-efficient load balancing in datacenters.

First, we introduce Nona, a stochastic scheduling framework that leverages queueing theory to optimize task placement in datacenter environments. By employing randomized algorithms and considering both network and compute constraints, Nona demonstrates multiple orders of magnitude improvements in job completion times while maintaining implementation simplicity. Nona proposes stochastic scheduling, in which the complexity of the scheduling problem is moved to an offline phase. When handling jobs online, stochastic schedulers are oblivious to the instantaneous state of the network and only rely on predetermined allocation probabilities to make lightning-fast decisions. Second, we present LINC, an in-network coding solution designed for hybrid backbone networks. Through comprehensive mathematical analysis and simulation, we highlight the benefits of network coding in cases where no modifications of the end-hosts are possible. Finally, we develop Sirona, a memory-efficient version of a reactive subflow spraying mechanism suited for hardware deployment. We show that Sirona can achieve competitive performance in homogeneous and heterogeneous datacenter networks while keeping a low memory footprint.

Thesis supervisor: Manya Ghobadi
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis supervisor: Muriel Médard
Title: NEC Professor of Software Science and Engineering

Acknowledgments

My PhD journey would not have been possible without the support and guidance of many remarkable individuals.

First and foremost, I am deeply grateful to my advisors, Professors Manya Ghobadi and Muriel Médard, for their unwavering support and mentorship throughout my graduate studies. Their expertise, insights, and encouragement have shaped both this work and my development as a researcher. I was extremely fortunate to have had the opportunity to learn and train by their side and will remember their teachings for the rest of my career and life.

I would like to express my sincere appreciation to Professor Mohammad Alizadeh for serving as my thesis reader and providing valuable feedback throughout my PhD that proved critical in strengthening this work.

I split my life at MIT between my two research groups, NCRC and NMS. My time in NCRC has been filled with memorable moments that I will always cherish. From therapeutic rant sessions about buggy results to entertaining game nights, this group has been a source of both academic growth and genuine camaraderie. A special mention goes to Kathleen for sharing helpful insights on life as a PhD student. The NMS group has provided a wonderful community, from day-to-day support to occasional ski trips, group hikes, and dinners offering a welcome respite from research challenges. Special mentions go to Sudarsanan, Mingran, and Frank who began their journeys alongside me during unprecedented times (Summer 2020, all working remotely). I have joked multiple times that they were too nice; all three of them, on top of their technical qualities, are incredibly generous persons who always sacrifice their time to help me and others. I truly appreciate your selflessness.

My parents and my brother have contributed to every step of this academic pursuit. Through exciting my curiosity, giving me the proper technical tools and knowledge to navigate both academic and non-academic worlds, and most importantly through their unconditional love and support, no part of this journey would have been possible without them.

To my friends, whether we met before I started this journey, or along the way, thank you for having provided much-needed levity and perspective during these times.

Special mention to my partner, who had the misfortune of meeting me towards the end of my PhD, in a time filled with tight deadlines and long work nights; in summary not enough time to spend together.

I would also like to acknowledge my collaborators, whose contributions and expertise have enriched this research significantly.

Finally, I would like to thank Thomas Clausen for providing a great introduction to academic research, to the point of convincing me to pursue a PhD degree.

This research was supported in part by NSF CNS-2008624, NSF SHF-2107244, NSF ASCENT-2023468, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, NSF FuSe-TG-2235466, and Sloan fellowship FG-2022-18504.

This Research was sponsored in part by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

This work was supported in part by the Center for Ubiquitous Connectivity (CUbiC), sponsored by Semiconductor Research Corporation (SRC) and Defense Advanced Research Projects Agency (DARPA) under the JUMP 2.0 program.

Published and Submitted Material

This thesis is built upon work published, under submission, or planning to be submitted.

Chapter 2 extends a paper published at IEEE CloudNet'2024 [1]:

> B. Pit-Claudel, D. Malak, A. Cohen, M. Médard and M. Ghobadi, "Nona: A Stochastic Congestion-Aware Job Scheduler for Real-Time Inference Queries," *2024 IEEE 13th International Conference on Cloud Networking (CloudNet)*, Rio de Janeiro, Brazil, 2024, pp. 1-10.

Chapter 3 was submitted at ACM CoNext'2025 and is currently under submission. The work was done in collaboration with Muriel Médard and Manya Ghobadi.

The core results of chapter 4 have been submitted to ACM SIGCOMM'2025. This work was performed in collaboration with Abdul Kabbani and Ahmad Ghalayini. It builds upon a chapter of Ahmad Ghalayini's thesis [2, chapter 5], and extends the solution and the analysis proposed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Network traffic has grown exponentially in recent years, driven by the convergence of multiple trends both within datacenters and across the broader internet. Within datacenters, the rise of Artificial Intelligence (AI) workloads and the ubiquity of Large Language Model (LLM) based solutions in modern industry products have led to unprecedented demands on network infrastructure [3–7]. Outside of AI workloads, modern microservice architectures multiply the number of network interactions needed to serve even simple user requests. Beyond the datacenter, the internet backbone faces its own explosion in traffic as businesses of all sizes shift their operations online, video streaming is now ubiquitous, and cloud services become the default choice for both enterprises and individuals [8, 9]. For all these applications, network latency is a critical Quality of Service (QoS) metric to minimize that must be tackled across the lifetime of packets and applications.

At the scheduling level, systems must make fast and complex decisions about task placement across thousands of machines. Current approaches often assume massively over-provisioned networks and focus only on efficient compute resource utilization. However, the network becomes the bottleneck for some new datacenters workloads, and cannot be ignored.

As datacenter scale increases, heterogeneous deployments require network load balancers to dynamically adjust to the network topology. Recent congestion control approaches have shown that in-order delivery of packets is not an indisputable necessity, opening the door for sub-flow-level load balancing methods. To see commercial deployments, these sub-flow-level load balancing solutions must be simple, well-understood, and sufficiently memory-efficient to be implemented in hardware.

Finally, outside of datacenters, backbone networks have seen the development of hybrid networks that rely on fiber, satellite, and microwave links to support ultra-low latency applications. While promising, these networks suffer from packet loss, eventually increasing effective End-to-End latency.

1.2 Problem Definition and Goals

This dissertation addresses three fundamental challenges in modern distributed network systems:

1. Inference Task Scheduling: How can we design scheduling systems that effectively consider both network and compute limitations? Current scheduling approaches often fail to account for networking constraints, leading to suboptimal performance.

2. Hybrid Networks Efficiency: How can we maximize hybrid backbone networks utilization and throughput for latency-sensitive applications?

3. Load Balancing: How can we achieve dynamic, resilient, and efficient load balancing across datacenter network topologies while minimizing memory overhead?

1.3 Contributions

This dissertation makes the following key contributions:

1. We introduce in chapter 2 Nona, a novel stochastic scheduling framework that uses queueing theory to determine optimal task placement probabilities. Our approach demonstrates the potential of using randomized algorithms for datacenter job scheduling, both in terms of performance and implementation simplicity. We develop a networking and compute model for inference jobs running on datacenter clusters to optimize placement probability distributions and show methods to minimize expected job completion time based on this model.

2. We present LINC in chapter 3, which shows the benefits of using an in-network coding solution with no assistance from end-hosts. We provide a comprehensive mathematical analysis of the system that matches our simulations to assist network operators in deploying LINC.

3. We develop Sirona, a memory-efficient load balancing system that achieves high load balancing performance while dramatically reducing memory requirements. Sirona is based on RSS [2], on which it improves by significantly reducing memory footprint while keeping competitive performances. We also extend RSS with a thorough analysis of the critical parameters affecting its performance and propose alternatives based on that analysis. At its core, Sirona combines insights from theoretical works on load balancing problems with heuristic-based approaches to extract the best of both worlds.

Chapter 2

Nona: A Stochastic Congestion-Aware Job Scheduler for Real-Time Inference Queries

2.1 Introduction

The compute and latency requirements of emerging online services, such as ChatGPT [10] inference requests, require dividing the workload across multiple datacenter servers [11]. As a result, job schedulers must efficiently distribute user-facing online Deep Neural Network (DNN) inference queries by meticulously considering computation resources, network capacity, and congestion while making fast real-time scheduling decisions.

To address this challenge practically, current job scheduling techniques rely on approximate representations of applications and deploy point solutions. They also depend on collecting real-time statistics about the datacenter, such as the status of currently running jobs or the instantaneous load of compute resources. However, *accurately measuring the queue occupancies of network switches in a cluster in real-time is nearly impossible* due to the bursty nature of datacenter traffic [12, 13]. Thus, the state-of-the-art job schedulers employ

heuristic-based approaches to focus on the compute requirements of jobs, either ignoring the impact of network congestion completely [14–16], fitting a network cost model from past executions [17–20], or relying on the job to provide its networking demand [21].

On the other hand, queueing-theoretic approaches have the potential to capture network congestion, but extending today's solutions to structured DNN jobs breaks the independence assumptions about the arrival processes of data transfers in the network queues. Therefore, applying them to datacenter workloads presents two significant challenges. First, the Directed Acyclic Graph (DAG) structure of today's online queries induces several locality dependencies. DNN's DAGs are composed of tens to hundreds of compute operations (or *tasks*) that use the outputs of their parent tasks as inputs; distributing two consecutive tasks on different compute resources introduces communication costs and requires carefully considering the impact of data dependencies between the tasks on network resources and queues. Second, successive tasks in a DAG introduce stochastic dependency in the arrival processes of individual tasks: the arrival process of the last task of a DAG depends on the service distributions of all the other tasks. Queueing models usually assume Poisson arrivals of jobs automatically result in Poisson arrivals of every task in each job's DAG. Given the depth of DAGs of today's DNN inference models, this assumption is not accurate. Therefore, today's queueing-theoretic approaches cannot be used out of the box for jobs with deep, complex DAGs. Prior papers avoid these challenges either by considering individual, isolated jobs [22] or by excluding DAG parallelism [23–25] and ignoring network congestion. To highlight the prohibitive complexity of network-aware scheduling, the recent AlpaServe [26] paper proposed a simple queueing-theoretic model to capture the communication overhead of scheduling inference queries. However, due to the complexity of their model, they fell back to a heuristic-based solution [26, Section 4.2].

We introduce in this chapter the concept of stochastic scheduling: every task to server allocation is decided randomly, by sampling from a predetermined probability distribution. Finding the right probability distribution can be done asynchronously from the operation of

the system (*e.g.* before the system is started), and updates to these probability distributions can be computed while the system is running. Stochastic scheduling has the benefit of allowing lightning-fast decisions at the scheduler, with a parallel, offline process solving the scheduling problem.

Based on stochastic scheduling, we propose Nona, a principled framework for distributing latency-sensitive inference jobs while considering *both compute and communication resources* of a cluster. Nona has three novel contributions. First, Nona uses an optimization formulation based on a flow network view [27, Chapter 3, 5.4] of the cluster that uses the well-established Pollaczek–Khinchine (PK) formula [28, 29] to predict network congestion and determine a series of stochastic decisions minimizing the average expected job completion time (JCT). Nona's optimization handles locality dependencies in the DAGs, building on prior work on networks of queues [30] to compute cluster settings.

Second, Nona's stochastic scheduler uses the solution found by the optimization to perform placement decisions in real time. This stochastic scheduler is lightweight, fast, and easy to implement (adding Nona's scheduler to the Apache Spark framework [31] requires ≈ 20 lines of Scala code): for each task in a job's DAG, Nona randomly samples a server from a pre-computed probability distribution obtained from solving the optimization.

Nona uses queueing theory to determine this series of stochastic decisions based on the steady-state properties of the workload. Consequently, Nona's optimization problem concentrates most of the complexity of our solution, leaving only lightweight operations to Nona's online scheduler. We argue that instead of developing complex online heuristic algorithms, latency-sensitive scheduling schemes should shift the complexity into an offline phase to enable making fast online decisions. This principle allows for solving the optimization and generating the corresponding probability distributions offline before running the system and adjusting them during operation without downtime.

Nona's optimization formulation leverages cloud providers' ability to continuously analyze traffic patterns *offline* and estimate each DAG's average arrival rate. In new deployments

where the arrival rate is unknown or volatile, cloud providers may solve Nona's optimization with a set of predicted arrival rates and adjust the predictions over time. Our evaluation in section 2.5.4 demonstrates that Nona is robust to changes in arrival rates of up to ±40%. Moreover, cloud providers may precompute multiple scheduling strategies corresponding to multiple arrival rate scenarios so that Nona's scheduler uses a lookup table that best matches the current arrival rate conditions. Overall, Nona requires strictly less live state information than state-of-the-art heuristics [14–21, 26, 32].

Finally, Nona introduces a novel graph contraction procedure to reduce the search space and complexity of the optimization formulation without changing the optimal solution. This graph contraction procedure enables Cloud providers to execute Nona's offline optimization formulation within a few seconds.

While this chapter focuses on inference queries, the methods developed here apply to any system handling structured queries submitted by a wide variety of independent customers. Inference queries of models that cannot fit on a single hardware accelerator are obvious candidates for Nona, since its main actionable lever relies on splitting up queries' tasks across different accelerators. Other inference models can however also benefit from the reduced memory usage on each single accelerator to increase batch sizes[33]. Beyond inference queries, other distributed computing protocols, rely on very low latencies while potentially benefiting from this divide-and-conquer model approach. For example, only providing a subset of all operations and data is a desirable property in web3 approaches, in particular for networking [34]. Furthermore, the concept of stochastic scheduling can be extended to other contexts such as hardware chips with tight scheduling deadlines. Overall, we propose in this chapter an example application to showcase the benefits of stochastic scheduling and provide the full corresponding solution.

To evaluate Nona, we implement our optimization problem and conduct extensive simulations comparing its performances to state-of-the-art schedulers such as Decima [16], and Spark [31], to an Expert scheduler, and to two custom lightweight congestion-oblivious sched-

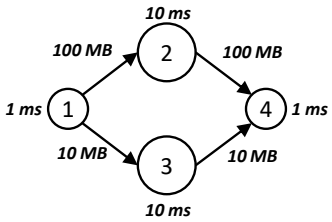
ulers (Random and Opportunistic). Using real-world DNN models (GPT-2 [35], DenseNet [36], VGG [37], AlexNet [38] and ResNet18 [39]), we demonstrate that Nona improves the average JCT by multiple orders of magnitude compared to Spark (56×), an Opportunistic scheme (180×), a Random scheme (202×), and Decima (350×), while being within 10% of an ideal Expert solution. Nona also improves 99%-*tile* tail JCT by 77×, 21×, 119×, and 81× respectively, compared to the same schemes. We then evaluate the impact of network bandwidth on Nona's performance and find that our results remain consistent for various network bandwidths. Finally, we study Nona's scalability and show that the execution time of solving our optimization problem is 30,000–245,000× faster than Decima's training time. Although both Decima and Nona have an offline phase, Nona is explainable, faster, and results in better performance.

2.2 Motivation

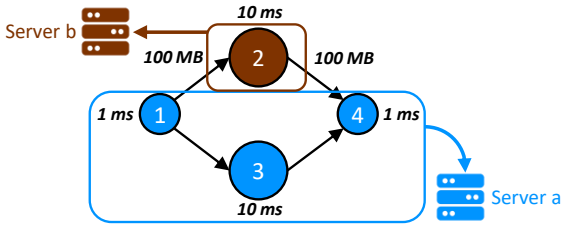
A significant part of current datacenter traffic increases is driven by ML workflows [19]. Newer DNN models (*e.g.*, recommendation models [40], large language models [10]) spend an increasing fraction of their runtime transmitting data on the network [41]. This is due in part to their sizes, which prevent users from running the entire model on a single accelerator, even for inference requests. For inference requests in particular, the end-to-end serving latency is critical. Therefore, minimizing network latency between distributed tasks in the DAG is equally critical.

2.2.1 Example

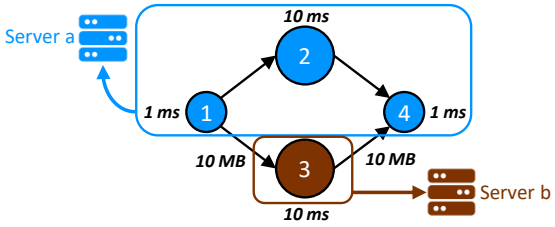
In this section, we explain how heuristic-based, congestion-oblivious schedulers can make poor scheduling decisions because of their inability to distinguish the network congestion implications of compute-equivalent distribution strategies. While we use a generic Congestion-Oblivious Schedulers in this section, our conclusions hold for other state-of-the-art scheduling



(a) A Sample DAG



(b) Allocation 1



(c) Allocation 2

Figure 2.1: (a) An example of a DAG with four tasks. (b, c) sample allocations. Tasks 2 and 3 are large and require 10 seconds to complete. Task 2 requires more data from the output of task 1 than task 3.

systems, such as Decima [16], Optimus [17], Gandiva [18], Tiresias [19], Pollux [20], Themis [21], Spark [31], INFaas [42], AlpaServe [26] and DRM-DQL [43].

Sample job

Consider a job f summarized by the following operations:

$$f : (A, B) \mapsto \det(A + B) + \det(r(A + B)) \tag{2.1}$$

Where for any square matrix M, $r(M)$ is a sparse function setting all elements of M to zero except for a single random one on each row, and $\det(M)$ is the determinant of M. Figure 2.1a shows the DAG corresponding to this job with four tasks: (1) Summing A and B, (2) Taking the determinant of that sum, (3) Taking the determinant of the sparse version of the sum, and (4) Summing the outputs of (2) and (3). The top branch of the DAG has larger transfer sizes than the bottom branch (all of $A + B$ has to be transferred instead of just a sparse version), and the middle tasks require more compute time than the first and last tasks (determinants

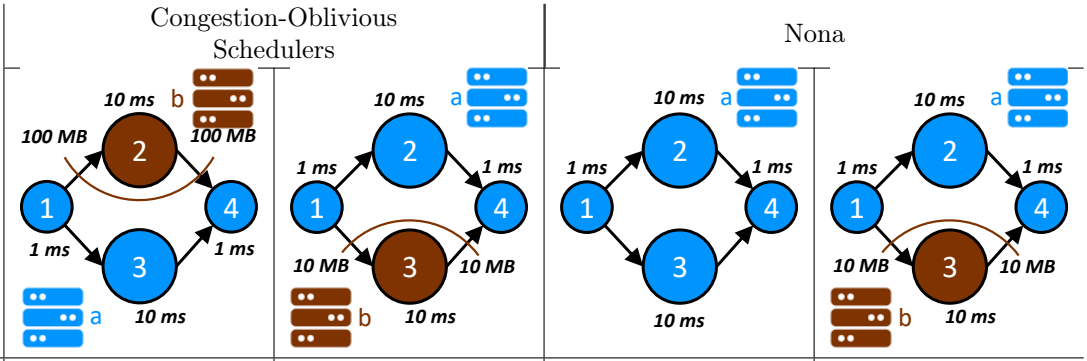
are more computationally expensive than sums). The duration of each task and the amount of data each task requires from the previous tasks are indicated next to the nodes and edges, respectively.

Single job scheduling

Consider a scenario with a cluster of two servers, a and b. We start by studying DAG distribution strategies for a single occurrence of f. Today's congestion-oblivious schedulers do not include any notion of topology or links between servers. When a job is distributed over multiple servers, it might pay a fixed penalty corresponding to various overheads, such as the startup time of a Java Virtual Machine (JVM) on that server. Moreover, congestion-oblivious schedulers tend not to employ cost functions that scale with the amount of data that needs to be transmitted between servers. Their outputs focus on the *number of servers* allocated to each task and the order in which different tasks should be executed. They are, therefore, unable to differentiate between two allocations with equivalent compute times but different network footprints.

As an example, Figures 2.1b and 2.1c show two possible task allocations by today's congestion-oblivious schedulers. Assuming servers a and b are equivalent, these two distribution strategies will have the same amount of compute on each server: place three tasks on one server (equivalent to $1 + 10 + 1 = 12$ ms compute time), and place the remaining task on the other server (10 ms compute time). However, allocation 1 in Figure 2.1b transfers 200 MB of data. On an 80 Gbps link, that represents 20 ms, or about 60% of the total JCT. In comparison, allocation 2 in Figure 2.1c requires transferring only 20 MB of data across the network corresponding to 2 ms on the same 80 Gbps link, or about 15% of the total JCT and 10× less than allocation 1. As a result, even though the compute scheduling of both allocations is equivalent, their corresponding network traffic is vastly different. In practice, today's congestion-oblivious schedulers end up randomly alternating between these strategies.

In contrast, Nona considers the size of data transfers on the DAG together with the



Cluster Load	Congestion-Oblivious Schedulers		Nona	
Low	50%	50%	20%	80%
Medium	50%	50%	50%	50%
High	50%	50%	80%	20%

Table 2.1: Allocation statistics for different schedulers.

available network bandwidth. As a result, Nona specifies which *groups* of tasks should run together on the same server instead of just the number of servers to allocate to each task. Consequently, Nona selects allocation 2 every time because it results in fewer bytes being transferred between the servers.

Scheduling streams of jobs

The previous example only considered a single job f. We now consider a more realistic case where a stream of users submits inference requests for f on different input data. State-of-the-art congestion-oblivious schedulers cannot distinguish between allocations 1 and 2. Thus, they end up load balancing between the two, as shown in Table 2.1. The same strategy is applied regardless of the resulting load on the link and the cluster's total network load. Instead, Nona captures queueing delays in its optimization formulation and makes scheduling decisions based on the load on the cluster. For instance, Nona sometimes decides not to distribute the job if distributing results in worse performance. As shown in Table 2.1, this decision depends on the expected average load of the cluster: when the network is lightly loaded, Nona chooses to distribute the computation and to use the network most of the time. When the network is busy, Nona mostly refrains from distributing f, to avoid adding to the network congestion and suffering from too much network delay.

This example highlights a significant potential source of gains over state-of-the-art congestion-oblivious schedulers. By incorporating network costs into the decision-making process, Nona is able to reduce network congestion and JCT.

2.2.2 Stochastic Scheduling

In this section, we argue for the benefits of making scheduling decisions based on pre-determined probability distributions instead of complex heuristics that consider instantaneous state information about the cluster. We showed in the previous section that past works avoided including networking in their decisions leading to potential performance hits.

In [22, Figure 3], authors present an optimization formulation for placement of training and inference workloads. That problem looks at the placement of a single job running in an isolated section of a cluster. The optimization variables used in that problem are therefore binary: for the considered job, either place a task on a server, or do not. On the other hand, in this chapter, we consider streams of jobs; a natural extension of that optimization problem therefore consists of considering stochastic variables. In practice, this means that the optimization problem answers the question "how often should a task be scheduled a certain way", instead of building a complex state machine.

From a modeling perspective, using stochastic variables enables a mean-field analysis of the system: if a task appearing every second on average has a 20% chance of being scheduled on a given node a, then in average a will run 20% of these tasks, according to the law of large numbers. As a direct implication, this means that any solution to the problem, whether feasible or not, whether optimal or not, yields an expected compute load and network demand.

From an implementation perspective, stochastic schedulers have the benefit of effectively displacing the complexity of the problem to an offline phase. When a new job arrives in the system, the scheduling decision consists of a series of samples from precomputed probability distributions. Practically speaking, that means Nona only takes a couple of lines of code to implement: for each request, load the corresponding distribution, sample, and assign the jobs

to the resulting servers. Determining these distributions is critical–as we show in section 2.5, but can be done offline. To show the potential of stochastic scheduling, we make the decision to include no instantaneous state information in the process: the choice of the distributions depends only on generic information about the cluster, and on specifics of the inference request's model.

One could argue that, with the proper lens, all schedulers can be analyzed as stochastic schedulers. Indeed, on a coarse timescale, running a scheduler on a wide range of scenarios, and grouping jobs of the same class together, we can draw statistics about the scheduling strategy used by the scheduler. However, as we show in section 2.5.8, summarizing a heuristic-based scheduler by its long-term statistical behavior loses too much information and yields both a different behavior and poor performance results. We therefore consider stochastic the schedulers that only rely on probability distributions to make decisions, or, put differently, we consider stochastic schedulers to be schedulers with an inherent random aspect.

2.3 Nona's System Description

In this section, we first provide some background on tools used in Nona's system analysis, then an overview of Nona's high-level design (§2.3.2) and relevant assumptions (§2.3.3). Then, we explain Nona's optimization formulation (§2.3.4). Finally, we describe two techniques to solve Nona's optimization for real-world jobs (§2.4.1).

2.3.1 Background

subsectionNona's Design Principles Nona is built upon the following theorems and assumptions.

Networks of Queues

Jackson first introduced Jackson networks in 1963 [30]. They consider a system where jobs or customers have to visit a set of service centers or servers in a given order. The set of service centers and the order is specific to each job. It is assumed that each customer visits service centers on their list, getting in line after arriving at a new service center—often in a FIFO manner. Arrivals of customers follow a Poisson process, and service times are exponentially distributed. By considering average transition frequencies between service centers, the system is summed up by the arrival frequency and a Markov process, with the number of customers in each queue serving as state, and the frequency matrix of transition from each pair of servers as transition probabilities. Jackson deduces the equilibrium distribution of queue occupancy. In the datacenter setting, however, jobs are not bound to specific service centers and can be served by multiple service centers at the same time according to their DAG. For this reason, we cannot directly use the equilibrium distribution.

[44] extends this model to multi-commodity flows. Multiple independent flows, each composed of a set of customers as in a Jackson Network, compete for the same communication resources; a result for the average waiting time spent by a packet in the network is derived.

Arrival and Service Assumptions

A similar result to the central limit theorem justifying the widespread appearance of the normal distribution exists for arrival processes.

Theorem 1 ([45, Proposition 11.2.VI]) *The superposition of a large number of independent sparse renewal processes leads to a Poisson process in the limit.*

As an example, in the case of a datacenter, customers submitting inference requests using the same model can be considered independent. In this chapter, we assume that the arrival process of each class of jobs (defined by its DAG) follows a Poisson process. We also assume that the service time of any task, both in compute and communication queues, is a constant:

the time variation between multiple executions of the same tasks is small compared to the runtime of typical datacenter tasks [46].

Kleinrock's Independence Assumption

Consider a simple Jackson Network with two service centers. Consider two subsequent customers c and $c+1$, and set time 0 as the time at which customer c arrives at the network. Let i_1 be the inter-arrival time at the first service center, and s_1 be the service time for customer 1 at service center 1. Consider finally i_2, the inter-arrival time between those two customers at the second service center. i_2 is lower bounded by $s_1 - i_1$ which can be strictly positive, *i.e.* the arrival process at the second service center is not Poisson.

In [47], Kleinrock shows that when the traffic in a queueing system is composed of a large number of *independent* users, each generating flows of *dependant* tasks, the inter-arrival time between packets of the compound flow follows an exponential distribution—*i.e.* the dependency between tasks of the same user is lost and arrivals at the second service center can be considered to be Poisson.

Pollaczek–Khinchine (PK) Formula

[28, 29] The PK formula gives the mean waiting time and mean time spent in an M/G/1 queue (a queue with a single service center or processing unit, Poisson arrivals, and general service time distribution).

Theorem 2 (PK) *The time W' spent waiting in the queue of the system before being served is a function of the first two moments of the service time distribution S and the arrival rate λ of the Poisson process:*

$$W' = \frac{\lambda \mathbb{E}\left[S^2\right]}{2\left(1 - \lambda \mathbb{E}[S]\right)}. \tag{2.2}$$

For multiple classes of clients, with respective arrival and service rates λ_j and S_j, we define $\rho_j = \lambda_j \mathbb{E}[S_j]$, the contributions to the system load from individual classes of clients, and $\rho = \sum_j \rho_j$, the total load of the system. In this chapter, the service time of compute tasks is considered constant for a given task, and we use $\mu_j = 1/\mathbb{E}(S_j)$ to denote the service rate of that task. Using [48], we get:

$$W' = \frac{\sum_j \lambda_j/\mu_j^2}{2(1 - \sum_j \lambda_j/\mu_j)} \ . \tag{2.3}$$

2.3.2 High-level Design of Nona

Nona consists of two components: first, an offline optimization formulation that minimizes the JCT of a series of jobs distributed in a cluster with stochastic allocation strategies. Nona's formulation uses the service provider's expected load of network and compute resources in the cluster to determine the expected queueing delays associated with different scheduling options. To do so, Nona uses the PK formula(see section 2.3.1) to compute the expected JCT corresponding to each job's DAG. This allows the formulation to capture the tradeoff between gains stemming from running tasks in parallel and additional network delay due to ensuing network congestion. The solution resulting from the optimization is a probability associated with each scheduling decision.

The second component to Nona is its stochastic scheduler that uses the solution found by the optimization to perform placement decisions in real time. Consequently, Nona's offline optimization problem contains most of the complexity of our solution, leaving only lightweight operations to Nona's online scheduler. Tasks are then executed on their chosen server in the order defined by the DAG.

2.3.3 System Model and Relevant Assumptions

To build our optimization formulation, we consider a stream of jobs to be executed on a cluster. Servers and links in this cluster are represented as queues with deterministic service times. Each server has a compute rate which corresponds to the number of compute operations it performs per second. We assume each server works on a single compute task at any given instant and has an infinite buffer to store pending tasks while it is busy. We divide the stream of jobs by class, where each class is defined by a DAG. In practice, for DNN inference jobs, each class represents a different DNN model.

We assume a large number of independent users submit inference requests to the cluster, thus following a Poisson arrival process [45, Proposition 11.2.VI]. Given that task completion times are not exponentially distributed, and given that jobs in datacenter settings are not bound to a single server and can be split across different servers, we cannot directly use results from Jackson Networks [30]. However, we use a similar flow network [49] approach and take advantage of the PK Formula to estimate buffer delays.

Today's datacenters interconnect all servers through a hierarchical topology, such as a Fat-Tree network [50]. Given that transport protocols adapt to the speed of the slowest link on the path regardless of the number of hops, we model the network topology with one bottleneck link. Our optimization problem can be extended to include network queueing delays on all links, but only yields marginal performance gains.

2.3.4 Optimization Problem

We start this section by outlining the high-level abstraction of our optimization problem, as follows:

Minimize:	Average job completion time (I)
subject to:	Communications, Computing, Flow, and Scheduling Constraints (II)-(XII) ,

(I) is the objective function to be minimized, (II), (III) and (VII) are the main constraints of this problem, and (IV)-(VI), (VIII)-(XII) define auxiliary variables, as we detail below. .

Let $\mathcal{J}$ be the set of job classes, defined by their operation DAG $\mathcal{G}_j$ (tasks $\mathcal{T}^j$, data dependencies $\mathcal{D}^j$). For any DAG $j \in \mathcal{J}$, let β_j be the average arrival rate of jobs with DAG j, and τ^j be the average JCT of all requests corresponding to job j.

(I) We define our objective function to be $\sum_{j\in\mathcal{J}} \beta_j \tau^j$. This objective function captures the average completion time of jobs, weighted by the arrival rate of each job class.

Stochastic scheduling.

A scheduling decision is a mapping of tasks onto servers, or equivalently a partition of the tasks in each DAG along with a mapping associating each part to a server. We frame our decision variables based on this graph partition view: we first divide the DAGs into subgraphs along bottleneck nodes (*i.e.*, nodes that are on every root-to-sink path) and then define probability distributions on partitions of these subgraphs.

More specifically, for a subgraph with source t_0, sink t_m, and intermediate tasks $T = \{t_1, \ldots, t_{m-1}\}$, let π be a partition of T, and let π_k and π_l be parts of π. Let Ω_t be the set of all possible triplets (π, π_k, π_l) for the subgraph with root t. Ω_t represents the set of possible mappings for the source and the intermediate tasks of the subgraph. Let $\theta_t[(\pi, \pi_k, \pi_l)]$ be a probability distribution on Ω_t. $\theta_t[(\pi, \pi_k, \pi_l)]$ corresponds to the probability of choosing mapping (π, π_k, π_l) where the subgraph's source is placed on the same server as tasks in π_k, the intermediate tasks are grouped according to π, and the subgraph's sink is placed on the same server as tasks in π_l. To illustrate these notations, we show in Figure 2.2b a sample partition π, with three parts, π_0, π_1, and π_2. The contracted root task (see section 2.4.1) corresponding to tasks $1 + 2 + 3$ is placed on the same server as tasks in π_2, *i.e.* tasks 12 to 15. Similarly, the sink task is placed on the same partition as tasks in π_1, *i.e.* tasks 7 and 11. This corresponds to choosing the mapping (π, π_2, π_1).

To completely characterize an allocation strategy, we must also determine the placement

of the roots of the DAGs. We notice that to obtain a Pareto-efficient solution, the average load of every server should be equal. Therefore we set the assignment of the roots of the jobs to uniform distributions.

The assignment distributions $\theta_t[(\pi, \pi_k, \pi_l)]$ are our optimization variables. They are valid probability distributions, hence:

(II) They sum up to 1:

$$\sum_{\Omega_t} \theta_t[(\pi, \pi_k, \pi_l)] = 1.$$

(III) And they are positive:

$$\theta_t[(\pi, \pi_k, \pi_l)] \geq 0.$$

Communication cost.

Let $b^{(t,t')}$ be the amount of information needed by each task t' from one of its predecessors t, and μ be the capacity of the bottleneck link. Let also $\mathcal{R}^j$ be the set of subgraph roots, $\mathscr{P}(t)$ and $\mathscr{S}(t)$ respectively be the set of predecessors and successors of t, and by extension $(t, t') \in \pi$ the event "t and t' belong to the same part of partition π". Each job arrival produces a set of communication arrivals, corresponding to edges in the DAG that are distributed over different servers. From job arrivals, we derive the average data transfer arrival and service rates (respectively $\lambda_{(t,t')}$ and $S_{(t,t')}$), and deduce a stability constraint on the link's load (ρ), the average queuing delay at the link (ϕ), and the average total communication delay ($c_{(t,t')}$):

(IV) Arrival Rates:

$$\lambda_{(t,t')} = \sum_{\Omega_t} \beta_j \theta_t[(\pi, \pi_k, \pi_l)] \cdot \begin{cases} \mathbb{1}(t' \notin \pi_k) & \text{if t}\in \mathcal{R}^j \\ \mathbb{1}(t \notin \pi_l) & \text{if t'}\in \mathcal{R}^j \\ \mathbb{1}((t,t') \notin \pi) & \text{o.w.} \end{cases}.$$

(V) Service time:

$$S_{(t,t')} = \frac{b^{(t,t')}}{\mu}.$$

(VI) Link load:

$$\rho = \sum_{j \in \mathcal{J}} \sum_{t \in \mathcal{D}^j} \sum_{t' \in \mathscr{S}(t)} \lambda_{(t,t')} S_{(t,t')}.$$

(VII) Stability condition:

$$\rho < 1.$$

(VIII) PK formula:

$$\phi = \frac{\sum_{j \in \mathcal{J}} \sum_{t \in \mathcal{D}^j} \sum_{t' \in \mathscr{S}(t)} \lambda_{(t,t')} S^2_{(t,t')}}{2\,(1-\rho)}.$$

(IX) Communication delay:

$$c_{(t,t')} = \phi + S_{(t,t')}.$$

Cost propagation.

Let p^t be the number of operations required by each task $t \in \mathcal{T}^j$ and ν the compute power of servers. Let $\mathcal{N}(t)$ the root node coming after t in $\mathcal{R}^j$, sorted in topological order. We independently compute the average completion times for each subgraph. First, we compute $\chi_{t,k,l}[(\pi, \pi_i)]$, the time required for all branches in part π_i of π to complete given that (π, π_k,

π_l) was chosen, for every partition. Each task's completion time is given by the sum of *(i)* its compute time and *(ii)* its communication time before and after itself, if any. Since all tasks in part π_i run on the same node, they cannot run in parallel and the completion of π_i is given by the sum of individual task completion times. Then we average those times to κ^t, the average time between the completion of the source and sink of a subgraph. Different parts run on separate servers, therefore the completion time of all the parts is given my the maximum of the completion time of individual parts. This formula could include another source of queueing delay for the access to a compute resource. We choose to omit it since this cost is paid once by every part and thus does not change the optimal assignment distributions while simplifying the optimization problem. Finally, we propagate the completion times of individual subgraphs to obtain the average JCT:

(X) Conditional branch completion time:

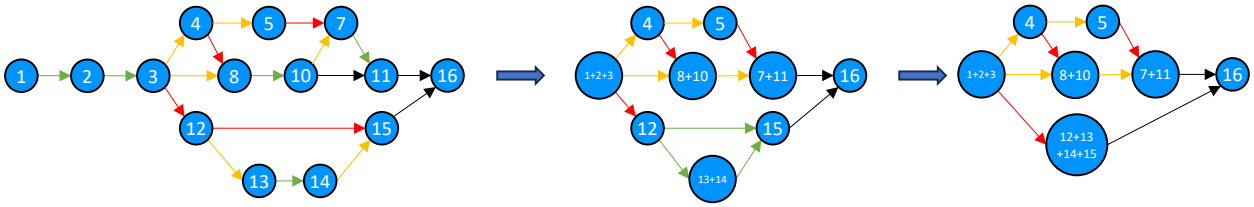
$$\chi_{t,k,l}[(\pi,\pi_i)] = \sum_{t_2\in\pi_i}\sum_{t_1\in\mathscr{P}(t_1)}\left[\frac{p^{t'}}{\nu} + \mathbb{1}(t_2\notin\pi_i)c_{(t_1,t_2)} + \mathbb{1}(t_1 = t\wedge i\neq k)c_{(t_1,t_2)}\right] + \mathbb{1}(i\neq l)\sum_{t_1\in\pi_i|(t_1,\mathcal{N}(t))\in\mathcal{D}^j} c_{(t_1,\mathcal{N}(t))}.$$

(XI) Subgraph completion time:

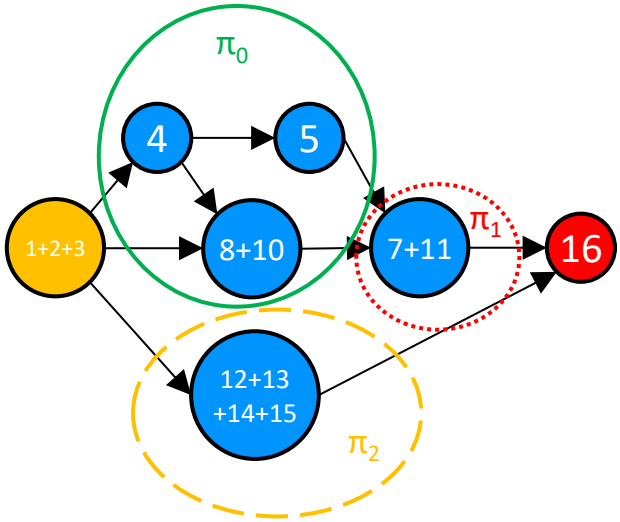
$$\kappa^t = \sum_{(\pi,\pi_k)}\theta_t[(\pi,\pi_k,\pi_l)]\max_{\pi_i\in\pi}\{\chi_{t,k,l}[(\pi,\pi_i)]\} + \frac{p^{\mathcal{N}(t)}}{\nu}.$$

(XII) JCT: $\tau^j = \frac{p^{t_0}}{\nu} + \sum_{t\in\mathcal{R}^j}\kappa^t$.

Due to equations (IV, VIII, XI), the objective can at most be reduced to a geometric fractional function in simple cases, hence Nona's optimization formulation is not convex. We discuss practical considerations to approximate the optimal solution in the following section.



(a) Contraction procedure.



(b) Graph partitioning.

Figure 2.2: (a) Contraction procedure on a sample graph. (b) Summary of graph partition notations.

2.4 Practical Solving Considerations

2.4.1 Search Space Reduction

Scheduling jobs while taking into account the network is not a convex problem. A simplification of Nona's problem on small DAGs yields a geometric fractional objective function, with equality constraints on posynomials; the full version is even more complex. We introduce two novel techniques to reduce the search space of the optimization formulation. Note that the last technique is already reflected in the formulation presented in section 2.3.4 through the choice of variables.

DAG Contraction.

First, we notice that the size of the search space scales with the number of tasks in the DAGs. Depending on the DAG, some tasks need to be executed sequentially. Consequently, executing sequential tasks on different servers brings no benefit since they cannot be run in parallel. We use this observation to *contract* edges with sequential endpoints into a single node.

However, contraction needs to maintain correct task dependency. For instance, only isolated edges—subsets of the graph limited to a single input and output—may be contracted. Otherwise, the resulting graph would not faithfully represent dependencies between tasks and would present unnecessary idle periods.

The following contraction procedure leverages this insight to keep only edges whose endpoint might gain from parallel execution: as long as the DAG can be further contracted, contract all edges $a \to b$ where all input nodes of b are also input nodes of a, and all output nodes of a are also output nodes of b. More rigorously, given $\mathcal{G}(\mathcal{T}, \mathcal{D})$ a graph to be contracted, with nodes $\mathcal{T}$ and edges $\mathcal{D}$, the set of edges to be contracted in a contraction step is given by:

$$\mathcal{C} = \Big\{(a,b) \in \mathcal{D} \mid \forall n \neq a, (n,b) \in \mathcal{D} \implies (n,a) \in \mathcal{D}, \\ \forall n \neq b, (a,n) \in \mathcal{D} \implies (b,n) \in \mathcal{D}\Big\}$$

Figure 2.2a illustrates this contraction procedure. Edges to be contracted in the next step are highlighted in green. Some edges seem contractable but are not; we highlight them in yellow and show the edges preventing them from being contracted in red. For example, contracting the edge between nodes 4 and 5 would lead to node 8 waiting for node 5 to complete before starting, a dependency not present in the initial graph. As shown by the bottom branches (nodes 12 to 15), multiple rounds can be necessary to ensure the graph is fully contracted.

Relative Assignments.

The second technique we use takes advantage of the relative assignments between tasks and servers. In particular, we argue that the critical question the optimization problem should answer is not which specific task should be placed on which specific server. Rather, the assignment decision can be reduced to a set of *relative* decisions: which tasks of a job should be grouped together on the same server.

Therefore, Nona's optimization uses $\theta_t[(\pi, \pi_k, \pi_l)]$ to group tasks together on an arbitrary server, instead of looking at which specific server should host a given task. Note that relative assignment makes the optimization formulation independent of the number of servers, thereby enabling Nona to scale to large clusters. Our evaluations demonstrate this highly desirable feature in Section 2.5.

Compute congestion

By a symmetry argument, we notice that all servers should have the same compute load. Indeed, any load imbalance between servers is not Pareto optimal: as shown by the PK formula, the queueing time for M/G/1 queues is supra-linear in the load. For example, in a cluster of two servers, an allocation with 51% of the load on the first server and 49% of the load on the second server could be strictly improved by switching 1% of the demand to the second server. The 1% added traffic will increase the queueing time at the second server less than the reduction at the first server.

In our case, ensuring equal load can be done by scheduling the first task of each job at random, according to the relative computation power of the servers in the cluster. In cases where all the servers have the same compute capacity, this corresponds to a uniform distribution of the first task.

2.4.2 Search Space Size Characterization

In this section, we derive a loose upper bound to the size of the sample space of Nona's optimization problem. The size of the problem is proportional to the size of the search space, thus the solving time depends directly on it. While large, the size of the search space Γ is tractable. The problem scales linearly with the number of roots and is independent of the number of servers in the cluster.

Consider a subgraph with root t, and b_t parallel branches between source and sink. The size of the sample space Ω_t for this subgraph is given by the number of pairs (π, π_k) where π is a partition of $[\![1; b_t]\!]$ and π_k is the index of the chosen part in π or -1 if no part is chosen. Let B_b be the number of partitions of a set of size b_t, also known as a Bell number. Thus, using [51]:

$$\begin{aligned}
|\Omega_t| &= \sum_{\text{partition } \pi \text{ of } [\![1;b_t]\!]} (|\pi| + 1) \\
&\leq B_b(b_t + 1) \\
&\leq \left(\frac{0.792 b_t}{\ln(b_t + 1)}\right)^{b_t+1} \\
&\leq \left(\frac{b_t}{\ln(b_t)}\right)^{b_t+1}
\end{aligned} \tag{2.5}$$

Sample space sizes in different subgraphs are independent of each other, thus:

$$\begin{aligned}
|\Gamma| &= \sum_{j \in \mathcal{J}} \sum_{t \in \mathscr{R}^j} |\Omega_t| \\
&\leq \sum_{j \in \mathcal{J}} \sum_{t \in \mathscr{R}^j} \left(\frac{b_t}{\ln(b_t)}\right)^{b_t+1}
\end{aligned} \tag{2.6}$$

2.4.3 Numerical Approximation of Optimal Solution

We initially attempted to solve Nona's optimization formulation using traditional gradient descent methods, as well as more recent first-order and second-order methods implemented in Keras [52] (Among others: Adagrad [53], Adadelta [54], Adam [55, 56], FTRL [57]), but these methods are not efficient to solve constrained problems. While these methods were developed with nonconvex Machine Learning tasks in mind, they do not deal with constrained problems well. In our case, they struggled with validity constraints: when constraints (II, III) are satisfied, the gradient of the corresponding penalty is equal to 0, and the overall gradient points towards decreasing all probabilities simultaneously. Adding line search [58] on top of the optimizers to avoid oscillating around the frontiers of the feasibility set was of no help.

We default to an explicit evaluation of the loss function on a subset of the feasible set for distributions with discrete probability masses. As shown in section 2.5, while this generates approximate solutions to the optimization results, the resulting distributions yield competitive scheduling policies. Monte Carlo approaches, or more elaborate searches of the feasible set could bring solutions closer to the optimum, but we leave their exploration and evaluation to future work.

2.5 Evaluations

2.5.1 Methodology

To evaluate Nona, we augment the event-based simulator in Decima [16] to become network-aware. The simulator takes the following input parameters: (1) a series of DAGs, each corresponding to a class of jobs, (2) an average arrival rate for each job, and (3) the cluster properties such as the number of servers, network capacity, and compute capacity per server. Upon every task's arrival or departure, the simulator places the subsequent available tasks on servers according to the scheduling policy of the evaluated scheme. Finally, the simulator

reports statistics about job and task completion times.

Workloads.

We consider five classes of inference jobs constructed from GPT2 [35], Densenet121 [36], VGG16 [37], AlexNet [38], and ResNet18 [39], for which we retrieve jobs' DAGs using PipeDream's [59] profiling tool. We then contract each graph as described in section 2.4.1. We report model characteristics in Table 2.2, including the number of tasks in each DAG before and after contraction. Our contraction procedure reduces the number of tasks by up to a factor of 107, reducing the search space by similar factors. Some models, such as Alexnet and VGG are reduced to a single node by our contraction procedure. This indicates that their operations graphs are sequential with no model parallelism opportunities.

Model	Size (MB)	Layers	# of tasks in the DAG	
			Pre-Contraction	Post-Contraction
GPT2	487	164	178	**40**
Densenet	31	369	429	**4**
VGG16	528	40	41	**1**
Alexnet	233	22	23	**1**
Resnet18	45	61	71	**10**

Table 2.2: Properties of DNNs used in evaluating Nona.

To simulate a realistic multi-tenant cluster, we generate a series of background compute and communication jobs. Each background compute job consists of a single task in its DAG, with zero communication demand. Each background communication job has zero compute demand. These jobs' arrival rates and service requirements are chosen to simulate a steady arrival of light background operations in a cluster. Empirically, we choose the ratio of inference-to-background arrival rates to be 100:1.

Compared Schemes.

We simulate the following schemes:

Decima [16]: Decima is a reinforcement-learning scheduler to make scheduling decisions

based on instantaneous information about the state of the cluster. The model is congestion-oblivious: its training environment does not consider network delays. We use a DNN model trained on SQL queries provided by the authors.

Spark's Fair Scheduler: this scheme shares the compute resources of the cluster fairly between all active jobs. It also requires instantaneous information about the state of the cluster. By default, Spark's Fair Scheduler prioritizes large jobs and starves smaller jobs by blocking all the servers allocated to a job until it completes. We modify it such that tasks release their resources when they are done.

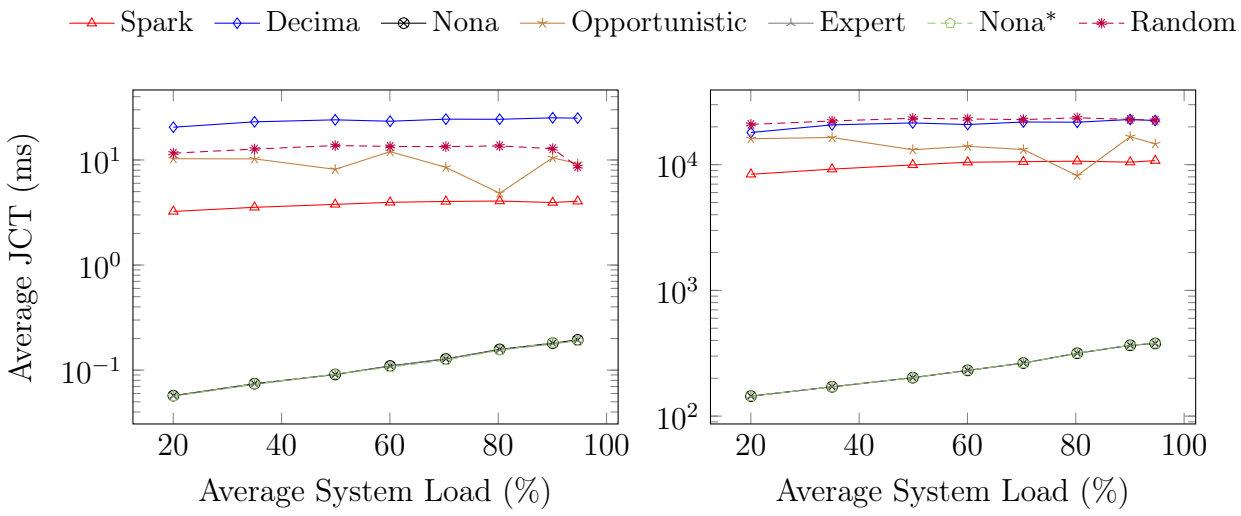
Random: Tasks allocations are chosen randomly. While this scheme has almost no overhead, it is both compute-oblivious and congestion-oblivious.

Nona: We run Nona's optimization offline separately from its simulator. The performances of our implementation of Nona's optimization are discussed in section 2.5.9. The output of the optimization is saved to a file. The simulator loads Nona's probability distributions from the file at startup, and uses them to make congestion-aware scheduling decisions.

Opportunistic: Whenever the contracted graph presents parallel branches, distribute all these branches on different servers. The specific servers are chosen randomly such that one of the branches is placed on the same servers as the root of the subgraph, making this scheme also congestion-oblivious.

Nona*: Nona's optimization problem takes the average arrival rate of jobs as input. In some cases where that information is not available initially, or if the arrival patterns change during operations, Nona would run with a non-optimal assignment strategy. To test Nona's robustness to varying system conditions, we run simulations using a single set of allocation probabilities, obtained from solving the optimization a single time, and use this same probability distribution for all the loads (or arrival rates).

Expert: this scheme uses a manually derived probability distribution. For every data point, we reason about the job's DAG and the system's expected load and determine the optimal distribution. This process is tedious and does not scale in the size of the cluster, the



(a) All jobs

(b) Inference jobs

Figure 2.3: Average JCT depending on system load.

number of DAGs, or the complexity of the DAGs.

Among these schemes, the Expert, Opportunistic, Random, as well as Nona and Nona* do not use the current state of the cluster queues when making scheduling decisions. Decima and Spark's Fair Scheduler use live state information about the state of the cluster and the completion progress of jobs to make decisions.

2.5.2 Nona's Overall Gains

Figure 2.3 compares different scheduling techniques' average JCT on a cluster with 80 servers and 10 Gbps link bandwidth. We vary the interarrival rates λ_j of jobs j and derive the corresponding compute load as $\sum_j \lambda_j S_j$ where S_j is the sum of the service times of all tasks of j.

As shown in Figure 2.3a, when considering all jobs in the cluster, Nona performs similarly to the Expert case, and outperforms Spark by a factor of 32 to 56×, the Opportunistic scheme by a factor 30 to 180×, the Random scheme by a factor 70 to 202× and Decima by a factor of 139 to 350× on average JCT. As the system load increases, the likelihood of having more active jobs than servers increases. Spark attempts to fairly allocate as many

servers to every job in the system. This means giving each job a single server. Thus, the average JCT for Spark plateaus as the scheduler stops performing any parallelism to maintain fairness. Nona still achieves lower JCT both average and 99%-tile (by a factor 77×), since even at high load, some amount of parallelism can be beneficial when low amounts of data are transmitted on the parallel branches. The JCT for the Opportunistic and Random schemes is dominated by network congestion even at low load, and therefore their performance is not affected significantly by load variations when compared with Nona's performance.

Similarly, Figure 2.3b shows that when the average is taken only over inference jobs, Nona also yields average JCTs similar to the Expert case and also outperforms Spark, the Opportunistic and Random schemes, and Decima by respective factors of 28 to 56×, 25 to 111×, 59 to 145, and 59 to 125×, respectively. Decima performs better when the average JCT is taken over inference jobs only since it prioritizes completing existing jobs over running the Shortest Remaining Tasks First (SRTF), and therefore starves the small background jobs.

The main reason for these improvements is network congestion: while Nona is offline and does not have an instantaneous view of the network, neither of Spark, Decima, or the Opportunistic or Random schemes take the network into account when making scheduling decisions. Therefore, all these schemes overload the network link by eagerly distributing jobs as much as possible, while Nona successfully prevents the system from jamming.

2.5.3 Impact of the Network Capacity

To isolate the impact of network resources on average JCT, we fix the arrival rate and vary both the capacity of the network and the network demand of each job, while keeping the compute load constant. In Figure 2.5, Nona still performs as well as the Expert solution, and yields consistent average JCT gains over a wide range of bandwidths. More specifically, Nona outperforms Spark's Fair Scheduler, the Opportunistic and Random schemes, and Decima, respectively by up to 22× (27×), 60× (44×), 70× (57×), 111× (44×) when the average is taken over all jobs (over only inference jobs). Here again, Decima suffers particularly from

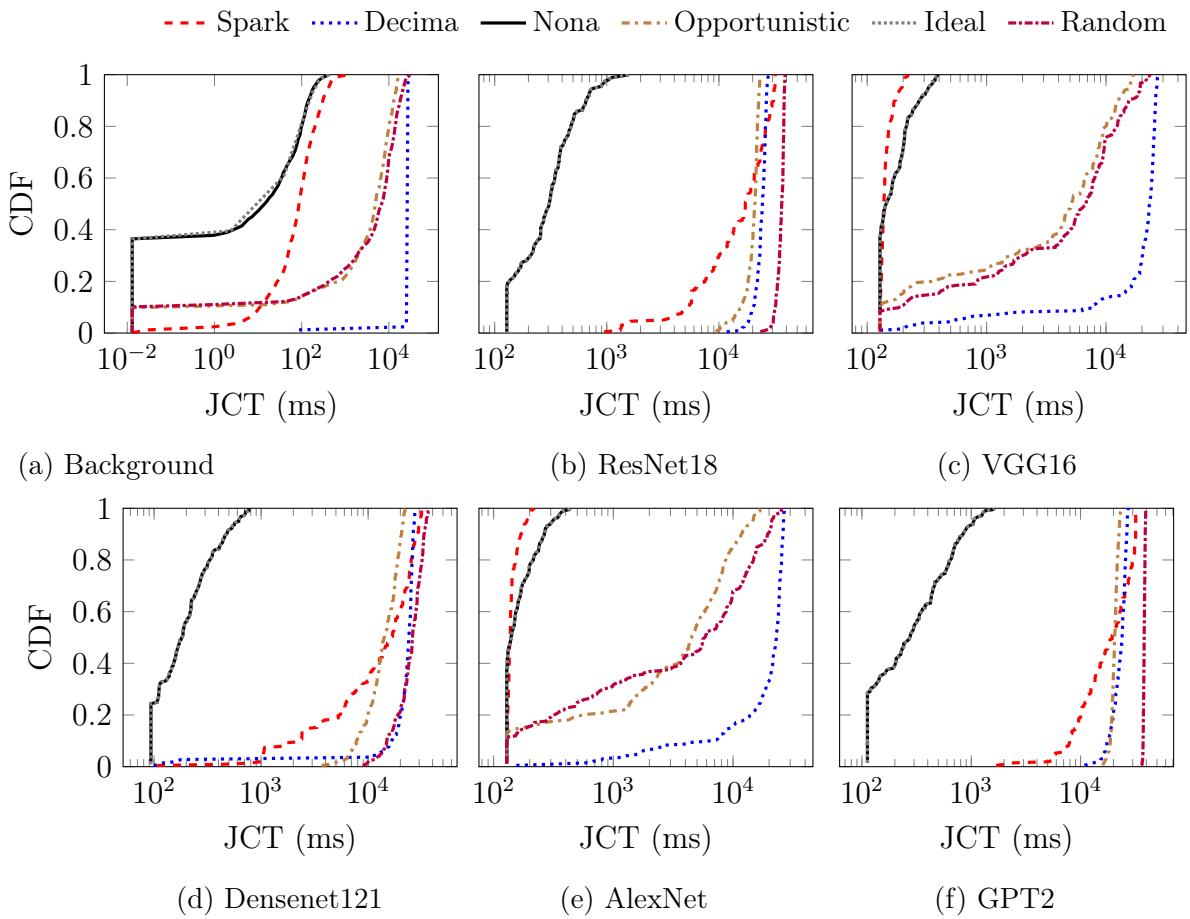
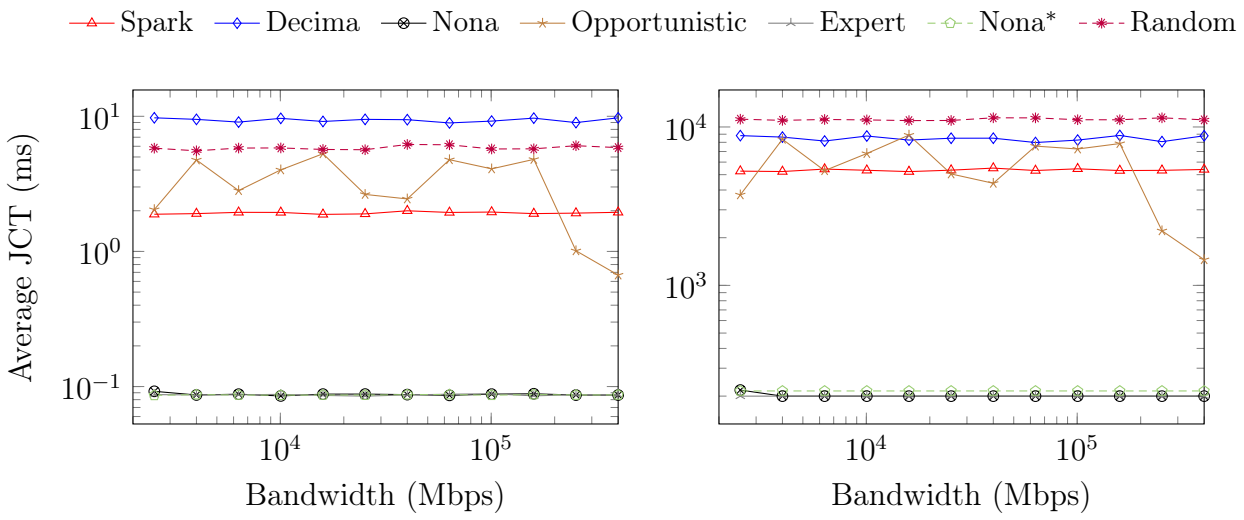


Figure 2.4: CDF of the JCT for each job, on a 80-server cluster, with 70 % communication and compute load.

starving background jobs.

2.5.4 Nona's Robustness to Arrival Rate Uncertainty

To show the robustness of Nona's optimization to uncertain arrival rates, we run the optimization a single time, for expected loads and bandwidths of 50 % and 10 Gbps respectively. Then, we run the same experiments as in sections 2.5.2 and 2.5.3 for Nona using only this single solution. The resulting JCT, labeled Nona* in Figures 2.3 and 2.5, show Nona does not require a precise knowledge of the effective arrival rates. Nona*'s performance is within 15% of Expert and Nona, and therefore outperforms all the other schemes by factors similar to Nona's.



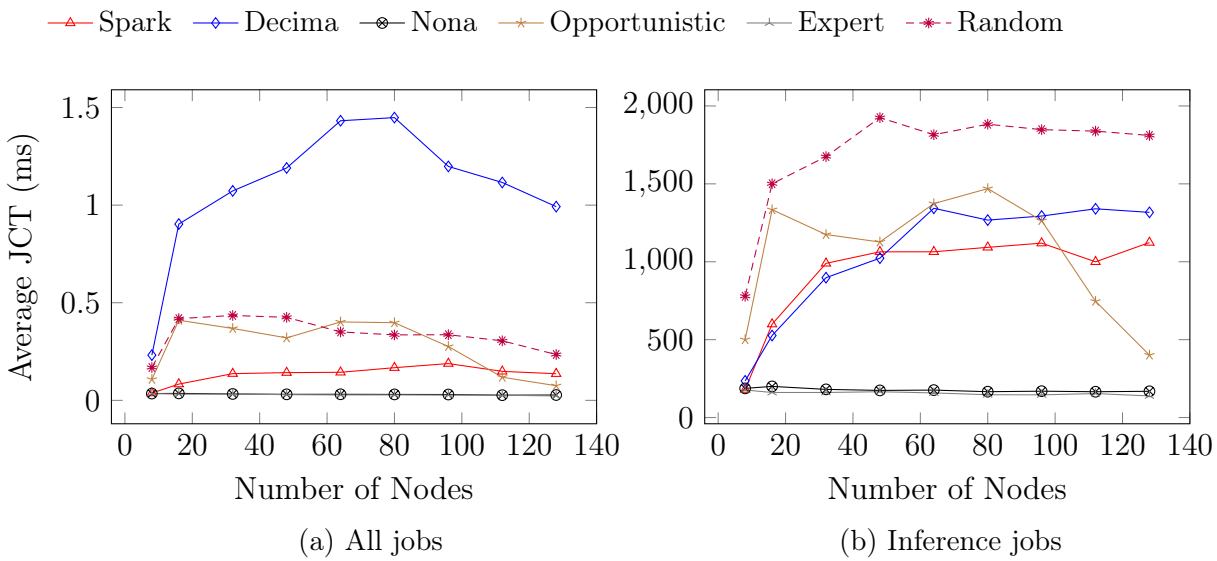
(a) All jobs

(b) Inference jobs

Figure 2.5: Average JCT depending on link bandwidth.

2.5.5 Impact of the Job Structure

Figure 2.4 breaks down Figure 2.3 and shows the Cumulative Distribution Function (CDF) of JCTs for each job class, at a load of 70 %. For inference jobs, Nona performs similarly to the Expert allocation and outperforms every other scheduling approach when the DAG presents parallelism options (Figures 2.4b,d,f), and outperforms all other schemes except for Spark for jobs with a single task (Figures 2.4c,e). Nona gives slight priority to background jobs given that its objective function is an average over the JCT of all jobs, including background. Indeed, while we modified Spark to prevent head-of-line blocking for these small background jobs, Figure 2.4a shows that Nona serves almost 40% of background jobs with no queueing delay, compared to 1% for Spark. Overall, for inference jobs, Nona accelerates the 99%-tile tail JCTs over Spark, the Opportunistic and the Random schemes, and Decima by 57×, 15×, 89× and 58× respectively. Figure 2.4a confirms Decima suffers from starving the short and frequent background jobs, with 99%-tile tail JCT 343× higher than Nona. A similar behavior is observed for single-task inference jobs as demonstrated in Figures 2.4c and e: Nona achieves a 99%-tile tail JCT 343× lower than Decima.



(a) All jobs

(b) Inference jobs

Figure 2.6: Average completion time as a function of the size in number of servers of the cluster. The compute demand is scaled to keep the compute load constant.

2.5.6 Impact of the Cluster Size

Figure 2.6 also demonstrates that Nona's performance stays consistent as we scale the number of servers, while other schemes make an increasing number of poor scheduling decisions. More specifically, Spark's Fair Scheduler and Decima achieve $40-80\times$ $(75-95\times)$ and $130-530\times$ $(3-50\times)$ worse average JCT when summing over both inference and background jobs (only over inference jobs). The system load is kept constant at 50%. As mentioned in section 2.3.4, Nona's optimization formulation is independent of the number of nodes in the cluster. The arrival rates of jobs are scaled across experiments to keep the load constant.

2.5.7 Breakdown of the Job Completion Time

We present in Figure 2.7 a breakdown of the average JCT by compute and communication processing time and compute and communication queueing delays. For each job, we add the time spent in queues, on links, or being processed by a server by individual tasks. Due to some computation and communication running in parallel, the total might be different from figure 2.3. For clarity, we clip the graph at 2s, to show some of the communication costs.

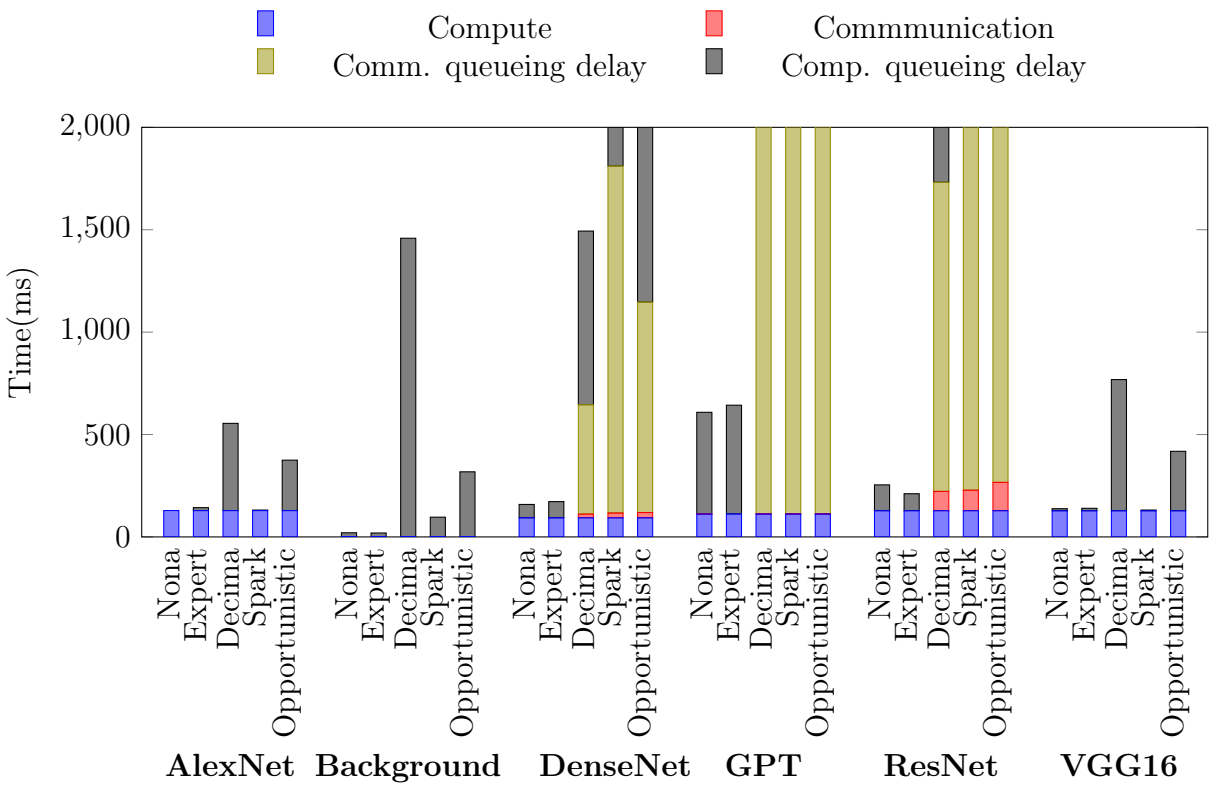


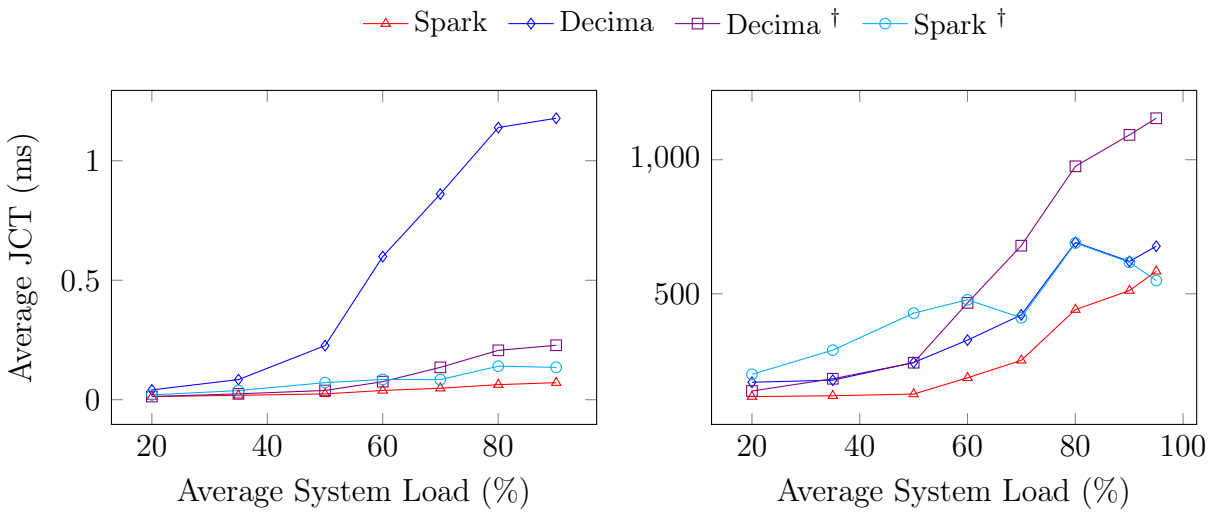
Figure 2.7: JCT split between computation and communication service and queueing times.

Within each job, the compute demand is similar across all the jobs, since the same operations need to be run. AlexNet, VGG16, and the background jobs are reduced to a single task after contraction, hence have no communication requirements, and no communication queueing delays.

Queueing delay plays a significant role in the final JCT. GPT is the only job for which the Expert decision and Nona choose to distribute some of the computation in this setting. The total communication cost is however negligible on the graph compared to other contributions and can therefore not be seen here. GPT, DenseNet121, and ResNet18, Decima, Spark, and Opportunistic all suffer from distributing computation too much.

2.5.8 Statistically similar stochastic schedulers

To measure the importance of the choice of distribution, we craft two additional baselines, Spark †and Decima †. In both cases, we first run an experiment with Decima or Spark, and



(a) All jobs

(b) Inference jobs

Figure 2.8: Comparison of Spark and Decima with their statistically similar stochastic schedulers.

collect placement statistics. More specifically, for each task in each job class, we measure the probability of running it on a server given the specific server that hosted its predecessors. We then run the same experiment a second time with a stochastic scheduler using the probability distributions collected by the prior run. To have meaningful statistics, we shrink the cluster size to 8 nodes and run 1000 total jobs in both cases.

We compare in Figure 2.8 the JCT for Decima, Spark, Spark †, and Decima †. In Figure 2.8a, Decima actively starves background jobs and therefore achieves a worse JCT than Spark †; Decima †performs slightly worse than Spark. In Figure 2.8b, both statistically similar stochastic schedulers perform slightly worse. This shows that the choice of the specific probability distribution has to come from a model of the system and that Spark and Decima need online information to function properly.

2.5.9 Nona's Optimization Implementation

Some linearization techniques have been proposed to solve problems with geometric fractional objective functions [60, 61] like Nona's. However, these methods consider problems with no

equality constraints, as opposed to Nona (equation II). Therefore, we implement the objective function of the Lagrangian relaxation of Nona's problem, and attempt to minimize it. The Lagrangian parameters are set empirically to multiple orders of magnitude above the expected JCTs, to sufficiently penalize points outside of the feasibility region. After contracting jobs, the size of the search space for the five jobs chosen is of 46, spread across 9 probability distributions; in general, the problem scales linearly with $\sum_j |\mathcal{R}^j|$ and is independent of the number of servers in the cluster.

Scheme	Number of servers							
	16	32	48	64	80	96	112	128
Decima	108h	215h	321h	438h	555h	651h	765h	889h
Nona	13s	13s	13s	13s	13s	13s	13s	13s

Table 2.3: Training/optimization time comparison.

Table 2.3 reports the runtime of our single-threaded implementation of Nona's optimization's solver, running on one AMD EPYC 7502P CPU, and compare it for various cluster sizes to the time required to train Decima's RL model on an Nvidia A100 using the same input parameters. We scale the training set proportionally to the size of the cluster to ensure a constant average number of jobs per server, explaining the linear scaling in training time. Nona, on the other hand, uses a queueing theoretic model independent of the cluster size to capture the essence of the problem's properties, and solves its optimization formulation 30,000×–245,000× faster. On top of this, our simple approach for solving Nona's optimization problem is fully parallelizable, and could be made multithreaded. Smarter approaches to solving the optimization problem could further reduce the time required by the optimization. We leave these improvements to future work.

2.6 Related Work

This section categorizes prior approaches into five classes of techniques: (*i*) Inference Schedulers, (*ii*) Heuristic-Based Schedulers, (*iii*) DNN Schedulers, (*iv*) Queueing-Theoric Tech-

niques, and (v) Optimal Resource Allocation.

Inference Specific Schedulers

Datacenter schedulers for inference tasks have gathered interest from the community in the past couple of years. However, as shown in the survey from Ye et al. [62], the focus of previous approaches has been on clever Machine Learning or GPU architecture based approaches to make the compute operations themselves more efficient. For example, in [63], the authors explored the benefits of caching data in GPU memory; in [32], the authors expanded on this idea by arguing for a system that only reloads the difference between two variants of the same model. INFaas [42] proposed a system to automatically choose model variants depending on the current cluster state. Finally, AlpaServe [26] demonstrated the benefits of model parallelism for statistical multiplexing of compute tasks in jobs. Our approach is based on the insight that the network must be taken into account for latency-sensitive online queries. Some of the techniques exposed in these papers (like caching) are compatible with Nona but would require some modifications of our optimization formulation; we leave this as future work.

Heuristic-Based Schedulers

Heuristic-based schedulers attempt to distribute tasks on a cluster by observing its current status and using a heuristic to make fast placement decisions. For instance, Gandiva [18], Tiresias [19], Themis [19], Pollux [20], and Pipedream [59], proposed several algorithms to distribute machine learning training jobs while optimizing compute utilization, compute throughput, or fairness metrics. However, these approaches did not consider *latency-sensitive* user-facing inference jobs. Sparrow [64] proposed an online randomized sampling mechanism to determine the status of compute queues before allocating tasks to servers. Yet, Sparrow's approach is not extendable to network queues because of the bursty nature of network flows [12]. In addition, the key difference between online schedulers and our approach is that

online schedulers rely on the short-term behavior of queues while our queueing theory-based approach draws on the long-term stochastic system characteristics.

DNN Schedulers

Recent DNN models have become sufficiently large to warrant dedicating entire clusters or isolated subsets of clusters to single jobs [41]. DNN-specific approaches thus work offline to derive an optimal allocation of resources specific to one cluster and one job. In [22], the authors derived an optimal DNN job DAG split minimizing the training time. In [15], the authors explored parallelization dimensions beyond model and data parallelism and found strategies combining them. PipeDream [59] and Gpipe [65] improved pipelining for DAG parallel training of a single job. Gpipe assumed that the entire cluster is available for a single job; consequently, Gpipe is not tailored to consider interactions between different jobs. Therefore, extending these solutions to handle *streams* of inference jobs is non-trivial: it requires considering much larger search spaces that consider *multiple DAGs* and their interactions.

Queueing-Theoretic Techniques

Unlike heuristic-based schedulers, queueing-theoretic scheduling techniques leverage the steady-state properties of the cluster to distribute a series of tasks on a network of queues [24]. For instance, Jackson networks [30] consider networks of queues with Poisson arrival processes and exponential service time distributions to summarize the flows of jobs by a Markov model [25], and to derive a product-form expression of the JCTs. However, Jackson networks consider independent customers, whereas datacenter jobs DAGs introduce *dependency* between tasks. On top of this, DNN inference workload runtimes do not match exponential distributions, and only approximations are available for Jackson networks with non-exponentially distributed service times.

Optimal Resource Allocation

Several papers have proposed scheduling methods for non-DNN jobs. For example, [66] derived optimal query execution plans for geo-distributed data. Similarly, [67, 68] studied latency-optimal scheduling schemes and proposed methods to optimize resource utilization. Finally, [69] explored dependency-aware scheduling. Yet, these works do not consider DAG parallelism for user-facing inference jobs with a complex DAG structure, often because they are geared toward smaller jobs with DAGs reduced to one task. On top of this, these solutions do not take into account *network congestion*.

Distributed coded computation.

Coded computing has been proposed as a way to reduce the communication requirements of tasks by adding redundant compute operations. As such, it circumvents communication bottlenecks by reconstructing results from a subset of successful tasks. For instance, [70–72] study the performance of coded computation in distributed computing for various tasks. Other approaches look at leveraging available data and compute redundancy [73–80] to perform coded computation. While these distributed techniques optimize the communication and computation tradeoffs, they are only applicable to linear functions or multivariate polynomials because of the decomposable structures of such functions. Thus, they cannot be easily extended to general nonlinear functions, which are required by ML models studied in this chapter.

2.7 Summary

In this chapter, we present Nona, a stochastic, queueing-theory-based scheduler for DAG parallelism in datacenter clusters. Nona uses an optimization formulation to derive placement probability distributions minimizing average job completion time. Our approach takes into account both network and compute service and queueing times, and can easily be extended to

consider other constraints (*e.g.*, memory). We show that Nona outperforms state-of-the-art heuristic-based solutions by up to 350×.

Chapter 3

LINC: An In-Network Coding Approach to Tame Packet Loss in Hybrid Wireless-Fiber Backbones

3.1 Introduction

End-to-End (E2E) latency and goodput play a critical role in the QoS metrics of ultra-low latency applications. Major internet service providers reported a direct correlation between latency increases and business revenue [81, 82]; for example, AWS reported in [83] a 1 % loss in sales per 100 ms of added latency. Conventional backbone networks rely heavily on fiber-optic deployments to carry both throughput and latency-sensitive applications.

However, fiber deployments depend on geographical constraints: laying fiber in some areas can be impossible or prohibitively expensive resulting in long detours and added latency. As a result, emerging ultra-latency-sensitive application providers deploy hybrid wireless-fiber backbones to carry traffic through a mix of fiber, satellite, and microwave links. For instance, in cISP [84], the authors demonstrated the competitive performance of a microwave link between Washington, DC, and New York City. Industry actors like Starlink [85], Taara [86],

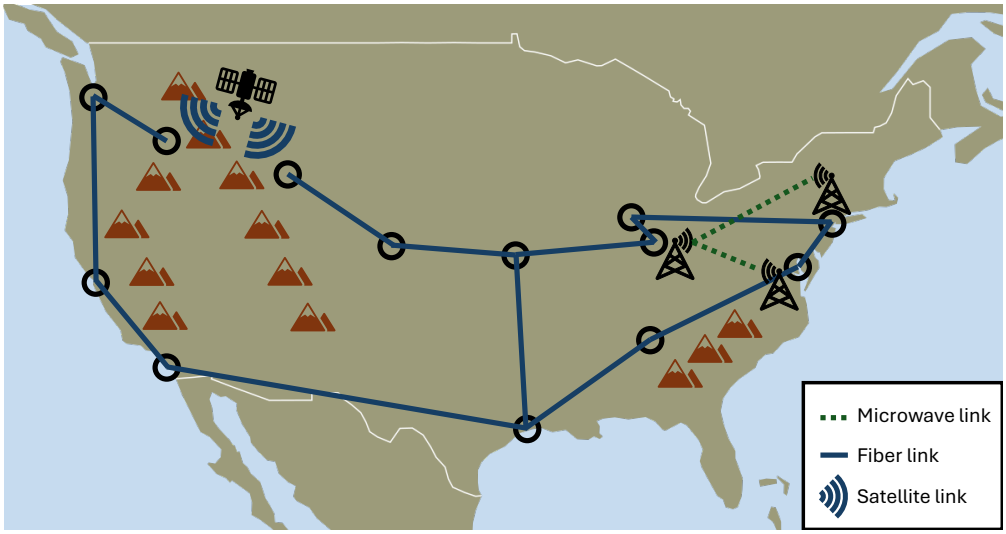


Figure 3.1: An illustration of a hybrid wireless-fiber backbone. The topology is inspired by Topology Zoo [95].

Kuiper [87], or TeleSat [88] have developed new commercial Internet Service Providers (ISP) serving customers through wireless links. On top of a more direct path, these alternatives also benefit from shorter propagation delays compared to fibers.

While these hybrid deployments outperform fiber-only backbones in optimal scenarios, they are subject to high environmental-induced packet loss: weather events [84, 89], physical objects blocking the line of sight path [90], and human-induced interferences [91] impact signal-to-noise ratio and result in transient packet loss surges [92–94].

Conventionally, end-hosts implement transport protocols to recover from network packet drops. The choice of the transport protocol is, therefore, agnostic to the types of links used in the backbone networks. Most transport protocols in today's backbones [96] treat loss as a signal for congestion. Therefore, environmental-induced packet loss translates into reduced sender rates and link under-utilization.

To address the above challenge, prior work proposed using Network Coding (NC) [97–100]. NC offers strong resiliency to random packet loss and provides low-latency guarantees. However, prior NC approaches assume access to the end-host network stack [101–104].

In this chapter, we argue that this assumption does not always hold, particularly in

hybrid wireless-fiber networks. We posit that sans access to end-hosts, encoding and decoding operations must be performed somewhere else on the path. Consequently, we propose a novel approach to tame the impact of environmental-induced packet loss on reliable congestion control protocols without requiring changes to the end-hosts.

Our system, called LINC, performs in-network NC while only assuming control over arbitrarily small subsets of the network switches. LINC relies on capabilities of programmable network switches, such as Intel Tofino [105], to implement a simple yet effective erasure coding mechanism inside network switches.

To perform NC, LINC uses a systematic block coding approach that we detail in section 3.3.1. Our theoretical analysis quantifies LINC's goodput gains by deriving the probability of retransmitting a packet with and without LINC. We quantify how the choice of coding parameters impacts the rate of retransmissions and the network goodput (§3.3).

We evaluate LINC in simulations on real-world backbone topologies and demonstrate that LINC eliminates packet retransmission due to environmental packet loss, therefore cutting E2E latency by up to 18 %.

3.2 Background and Related Work

Satellite and millimeter wave links. Several ongoing industrial efforts are underway to provide connectivity in areas where fiber deployments are challenging. For example, Facebook connectivity proposed Magma [106] as an effort to improve connectivity of remote rural communities, and TerraGraph [107] for last-mile connectivity in dense urban areas through millimeter wave links. Similarly, Taara [86] proposes beam-of-light communications for high-speed, high-capacity connections in areas where fiber deployments are not economically viable. Satellite-based commercial ISPs have existed for some time (see for example HughesNet [108] and ViaSat [109]) but suffered from high latency due to the altitude of their orbit. Recent low-earth orbit constellations like Starlink [85], Kuiper [87], or Telesat [88] report E2E

latencies in the order of tens of ms.

Environmental-induced packet loss. Environmental-induced packet loss in backbone networks is fundamentally linked to the physical vulnerabilities of optical and hybrid systems. Ghobadi et al. identified that backbone networks are highly susceptible to environmental disruptions, such as temperature fluctuations, which degrade signal quality over time, and physical incidents like fiber cuts or conduit damage during storms, leading to outages that cascade across channels sharing the affected segment [94]. Singh et al. observe that variations in signal-to-noise ratio, even under controlled conditions, can be amplified by external factors like weather events and infrastructure aging, causing significant signal degradation [92, 93]. Recently, Myers et al. showed that transient phenomena like rain, fog, atmospheric interference, and unplanned maintenance further exacerbate the instability of microwave-based links [89]. These environmental impacts often trigger packet loss spikes by crossing critical signal-to-noise thresholds, making recovery challenging and highlighting the inherent fragility of hybrid wireless-fiber networks to environmental-induced packet loss events.

Congestion control. Traditional E2E congestion control algorithms, like TCP, rely on packet drops occurring at full network buffers as a signal for congestion. This approach is based on the assumption that only congestion is responsible for dropped packets, which is reasonable for wired links with low erasure probability. For paths containing a wireless link, however, since TCP is unable to discriminate between packet loss due to channel erasures or due to congestion, the channel capacity will often be significantly underestimated. This shortfall of TCP in high-loss systems has been well-studied by the community [110–114].

Network coding. Among possible approaches, NC has shown to be an efficient solution to the above problem. First introduced as a capacity-achieving technique [115], NC has been extended and improved in various ways [98, 99, 104], providing high-bandwidth and low latency guarantees. In [101], authors have shown how to augment the TCP/IP stack with NC by adding a coding layer. This proposal also implemented re-encoding at intermediate nodes, providing flexibility to variations in the erasure probability on different links. In [97], authors

demonstrated the feasibility of NC approaches in software-defined networks and showed again the benefits of re-encoding at intermediate nodes.

Overall, these proposals have demonstrated the potential of NC. However, they all rely on access to and control of end-hosts. While encoding and decoding packets in-network has a latency cost compared to other E2E NC approaches, we argue in this chapter that NC should also be used in cases where the network provider does not have access to end-hosts and can only modify subsets of the backbone network.

3.3 System Design and Coding Approach

In this section, we first present LINC's design and assumptions (§3.3.1), before deriving the probability of retransmitting a packet in LINC and its impact on aggregate packet arrival rate (§3.3.2).

3.3.1 LINC System Design

Figure 3.2 illustrates LINC's system model. We consider a set of hosts sending data through a network to a set of receivers. Among them, senders h_1, h_2, ..., h_m share a lossy link, l, on their path to their corresponding receivers g_1, g_2, ..., g_m. For any sender h_i, we define η_i to be the number of non-lossy links on the (h_i, g_i) path. We model link l as a packet erasure channel [116], with an associated environmental loss probability ϵ. We assume that losses are not correlated.

Each source h_i generates packets according to a Poisson process with an average of λ_i packets per second. Let s_1 and s_2 be switches connected by link l, and consider the case where traffic flows from s_1 to s_2. In practice, acknowledgment packets traverse the other way; our analysis extends naturally to reverse traffic as well.

We take the viewpoint of a network operator, who only has control over some parts of the network, including s_1 and s_2, but not over users' devices or applications running on these

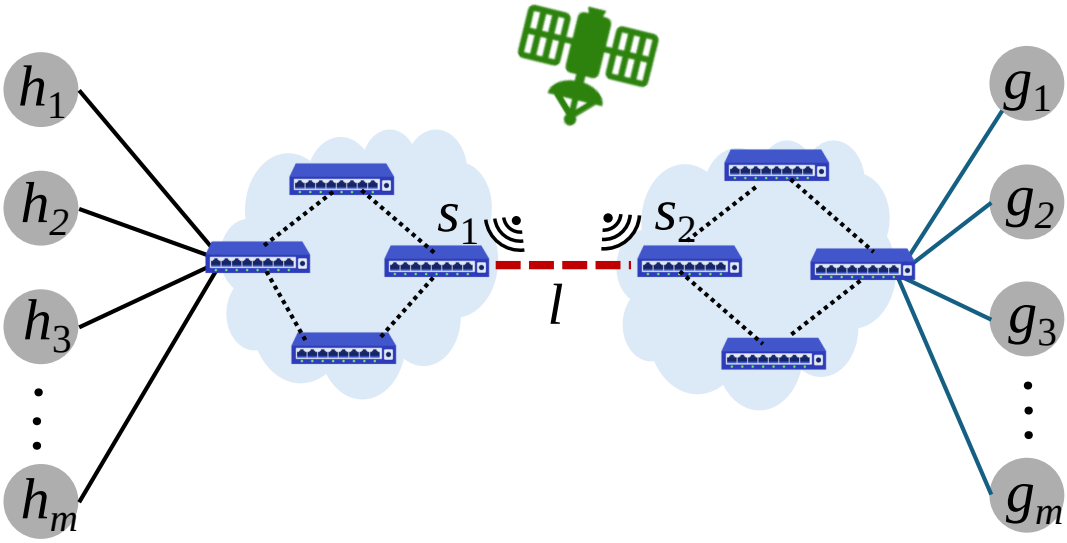


Figure 3.2: Illustration of LINC's system model.

devices. As these applications set transport protocols, we assume that traffic uses some version of TCP [96] that we do not have control on. Note that despite this assumption, our method is transparent to end-hosts, and therefore compatible with any other E2E coding protocols.

In-network encoding. LINC uses a systematic block code to encode packets at s_1. More specifically, for every block $B = (P_1, P_2, ..., P_k)$ of k consecutive incoming packets to s_1 with s_2 as the next hop, s_1 first transmits all packets of B unmodified. Then, s_1 generates $n - k$ additional packets destined to s_2, thereby sending a total of n packets to s_2. The process of creating the coded packets in s_1 consists of calculating a linear combination of packets in block B, where the coefficients are chosen such that if any k out of the n packets are successfully transmitted, the resulting matrix will be invertible (*i.e.* a maximum distance separable code [117]). LINC's packet header includes $\log_2(n)$ bits to identify the index of the packet in the block. This allows switch s_2 to determine whether a packet has been coded, and if so which linear combination should be used to decode it.

In-network decoding. Because of the environmental losses on link l, switch s_2 receives a subset of the n packets sent by s_1. Switch s_2 forwards successfully received uncoded packets

to their next hops without waiting for all the packets in the block to be transmitted. For every block, s_2 counts the number of successfully received packets. If at least k packets are received, s_2 decodes the block and starts sending the lost packets that it recovered. If less than k packets are received, s_2 moves on to the next block, so that the transport layer reliability mechanism can detect losses and trigger retransmissions.

3.3.2 LINC Theoretical Analysis

Consider the backbone network shown in Figure 3.2, where link l drops packets with probability ϵ. As discussed in the previous section, LINC recovers lost packets using its in-network encoding/decoding techniques. As a result, a LINC backbone network has a higher goodput compared to a backbone without LINC. In this section, we derive mathematical formulations to quantify LINC's goodput gains.

Retransmission Rate

In the block coding approach described in section 3.3.1, a block is successfully decoded if at least k out of n packets in the block are successfully transmitted. Given that our block code is systematic, when a block is not recoverable all lost uncoded packets must be retransmitted. Note that uncoded packets correspond to the original data packets and need to be retransmitted, whereas we do not need to retransmit coded packets that are lost. Let Q be the random variable corresponding to the number of losses in a block, and C be the random variable corresponding to the number of lost uncoded packets.

To quantify LINC's goodput gains, we start by deriving the expected packet retransmission rate R_{LINC}, by formulating the expected value of the number of uncoded packets that have been lost (i.e., $\mathbb{E}[Q, C]$), as follows:

$$
\begin{aligned}
R_{LINC} &= \frac{\mathbb{E}[Q, C]}{k} \\
&= \frac{1}{k} \sum_{q=n-k+1}^{n} \sum_{c=q-(n-k)}^{\min(k,q)} c \cdot \mathbb{P}(Q = q)\mathbb{P}(C = c \mid Q = q) \\
&= \frac{1}{k} \sum_{q=n-k+1}^{n} \sum_{c=q-(n-k)}^{\min(k,q)} c \cdot \binom{n}{q} \epsilon^q (1-\epsilon)^{(n-q)} \frac{\binom{k}{c}\binom{n-k}{q-c}}{\binom{n}{q}} \\
&= \frac{1}{k} \sum_{q=n-k+1}^{n} \sum_{c=q-(n-k)}^{\min(k,q)} c \cdot \binom{k}{c}\binom{n-k}{q-c} \epsilon^q (1-\epsilon)^{(n-q)} \\
&= \frac{1}{k} \sum_{q=n-k+1}^{n} \epsilon^q (1-\epsilon)^{(n-q)} \sum_{c=q-(n-k)}^{\min(k,q)} c \cdot \binom{k}{c}\binom{n-k}{q-c} \qquad (3.1)
\end{aligned}
$$

In the above equation, R_{LINC} is computed by taking the expected value of the joint distribution (Q, C). Using the law of total probability, we express this joint distribution as a function of Q and the conditional distribution of C, conditioned on Q. Since packet losses are independent, Q follows a binomial distribution. To determine the distribution of C, we draw the following analogy: in a block of n packets, there are k uncoded packets and $n - k$ coded ones. We randomly select q packets and try to determine the number of uncoded packets among these q. As a result, $(C \mid Q)$ corresponds to a hypergeometric distribution.

Aggregate Packet Arrival Rate

Next, we formulate the aggregate packet arrival rate over all links in the network. To do so, we first formulate the aggregate arrival rate of packets on link l (λ_{LINC}) by considering the following three categories of traffic on link l: (i) the aggregate raw traffic from all the senders ($\sum_{i=1}^{m} \lambda_i$), (ii) added coded packets ($\frac{n}{k}$), and (iii) the number of retransmissions for lost packets that were not recovered. The probability of a packet being retransmitted s times is given by R^s_{LINC}, thus the fraction of retransmitted packets is given by the sum of R^s_{LINC}

over the number of retransmissions. Putting it together, we have:

$$\lambda_{LINC} = \left(\sum_{i=1}^{m} \lambda_i\right) \cdot \left(\frac{n}{k}\right) \cdot \left(\sum_{s=0}^{+\infty} R_{LINC}^{s}\right) = \frac{n \sum_{i=1}^{m} \lambda_i}{k(1 - R_{LINC})} \tag{3.2}$$

Second, we formulate the aggregate arrival rate of packets on all the other non-lossy links in the network (λ'_{LINC}). As we are summing over all non-lossy links, the traffic from each sender contributes to the arrival rate of η_i links, hence:

$$\lambda'_{LINC} = \left(\sum_{i=1}^{m} \eta_i \lambda_i\right) \cdot \left(\sum_{s=0}^{+\infty} R_{LINC}^{s}\right) = \frac{\sum_{i=1}^{m} \eta_i \lambda_i}{1 - R_{LINC}} \tag{3.3}$$

The total aggregate packet arrival rate on a LINC network is $\lambda_{LINC} + \lambda'_{LINC}$.

Goodput Gains

Finally, to derive LINC's goodput gains, we first repeat the above calculations for the case without LINC. Then, we derive the ratio of goodputs for systems with and without LINC.

In a network without LINC, the retransmission rate is given by $R_{noNC} = \epsilon$, yielding a total arrival rate of packets on the lossy link, as:

$$\lambda_{noNC} = \left(\sum_{i=1}^{m} \lambda_i\right) \sum_{s=0}^{\infty} \epsilon^s = \frac{\sum_{i=1}^{m} \lambda_i}{1 - \epsilon} \tag{3.4}$$

Similarly, on the other links:

$$\lambda'_{noNC} = \left(\sum_{i=1}^{m} \eta_i \lambda_i\right) \sum_{s=0}^{\infty} \epsilon^s = \frac{\sum_{i=1}^{m} \eta_i \lambda_i}{1 - \epsilon} \tag{3.5}$$

Goodput is defined as the ratio of useful packets (the raw traffic from all senders traveling over $h_i + 1$ links for sender i) over the total number of packets transmitted. The goodput for

a system using LINC, $\mathcal{G}_{LINC}$, (respectively $\mathcal{G}_{noNC}$ for a system not using LINC) is given by:

$$\mathcal{G}_{LINC} = \frac{\sum_{i=1}^{m} \lambda_i (h_i + 1)}{\lambda_{LINC} + \lambda'_{LINC}} \qquad \mathcal{G}_{noNC} = \frac{\sum_{i=1}^{m} \lambda_i (h_i + 1)}{\lambda_{noNC} + \lambda'_{noNC}}$$

Finally, we define Δ, the ratio of goodputs for a system with LINC over a system without LINC:

$$\begin{aligned} \Delta &= \frac{\lambda_{noNC} + \lambda'_{noNC}}{\lambda_{LINC} + \lambda'_{LINC}} = \frac{\frac{\sum_{i=1}^{m} \lambda_i}{1-\epsilon} + \frac{\sum_{i=1}^{m} \eta_i \lambda_i}{1-\epsilon}}{\frac{n \sum_{i=1}^{m} \lambda_i}{k(1-R_{LINC})} + \frac{\sum_{i=1}^{m} \eta_i \lambda_i}{1-R_{LINC}}} \\ &= \frac{1 - R_{LINC}}{1 - \epsilon} \frac{\sum_{i=0}^{m} \lambda_i \left(\eta_i + 1\right)}{\sum_{i=0}^{m} \lambda_i \left(\eta_i + \frac{n}{k}\right)} \end{aligned} \tag{3.6}$$

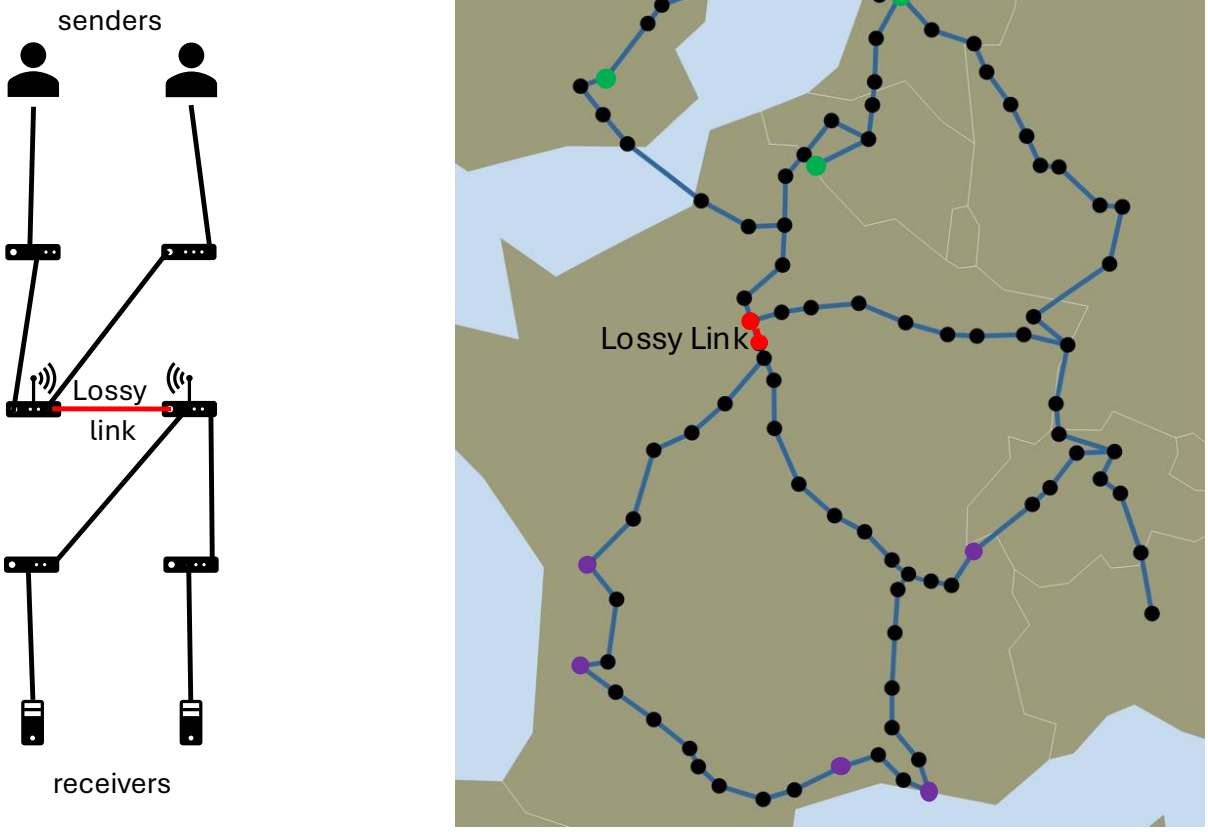
To determine the optimal choice of coding parameters (k, n) for LINC, we propose an optimization formulation that maximizes Δ subject to the following constraints: (i) $n \geq k$ and (ii) $k > 0$. The optimization takes the network topology, end-hosts sending rates λ_i and $\epsilon \in [0, 1]$ as input parameters.

3.4 Evaluations

To evaluate LINC, we implement a custom event-based simulator. This section describes our simulation methodology and compares its performance with the state-of-the-art.

3.4.1 Methodology

Simulator. Our simulation generates packets on senders following Poisson processes. The sending rates are randomly selected across senders and are normalized to an average network load of 50%. To simulate environmental packet loss, we select a single link to randomly drop packets. Each link is modeled by a shared resource processing packets. When a packet is lost, we trigger a retransmission after three duplicate ACKs are received by the sender. We



(a) Scenario 1

(b) Scenario 2: VtlWavenet2011 [95].

Figure 3.3: Topologies used in our simulations.

measure several parameters including the number of lost packets that were recovered using LINC, the number of retransmitted packets, and the arrival rate of packets at each link. We define the E2E packet delay as the difference between the time a packet was received successfully on a receiver, and the time the same packet was created at the corresponding sender.

Topologies. We consider two topologies, shown in Figure 3.3. In the first scenario (Figure 3.3a), two senders are sending traffic over a 5-hop network, each to a different receiver. The middle link is lossy, with latency 1 ms and drop probability 5%, unless otherwise stated. The latency for each non-lossy link is set to 100 ms. The number of non-lossy links for either sender is 4, as shown in Figure 3.3b. The second scenario (Figure 3.3b) is the VtlWavenet2011 topology taken from the Topology Zoo dataset [95]. We set a central link near Paris to be the

lossy link and choose the following (sender, receiver) pairs: (Marseille, London), (Bordeaux, Amsterdam), (Geneva, London), (Sete, Antoing), and (Blanzay, Strasbourg). This results in around 20 non-lossy links for each (sender, receiver) pair. The latency for each link is computed from the geographical distance between nodes. In both cases, all non-lossy links are assumed to have a rate of 100 Gbit/s. The lossy link is shown in red in both figures, and we highlight senders in purple and receivers in green in Figure 3.3b. For both scenarios, we set $k = 50$ unless otherwise noted.

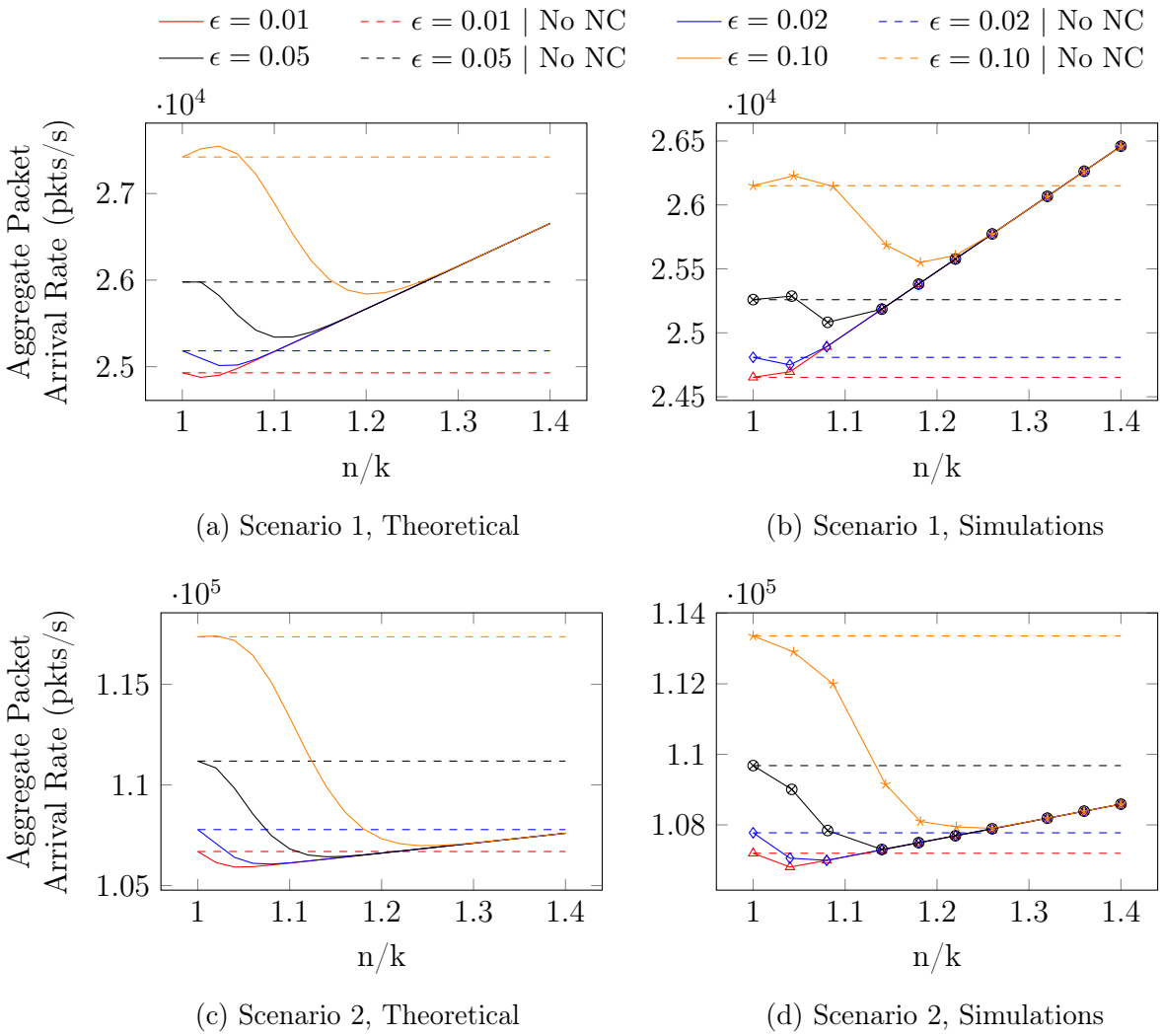
3.4.2 Aggregate Packet Arrival Rate

In TCP, retransmissions happen E2E. Therefore, when a packet needs to be retransmitted, it utilizes bandwidth on every link on its path. Figures 3.4 and 3.5 compare the aggregate packet arrival rate of LINC under different scenarios. In Figures 3.4a and 3.4b (respectively Figures 3.4c and 3.4d), we show that empirical results from our simulations closely align with our theoretical formulation (§3.3.2). The dotted lines represent the case without LINC.

As expected, the minimum value for the aggregate packet rate is a function of the packet loss rate ϵ. As ϵ increases, the amount of redundant packets needed to correct lossy transmissions increases. For instance, for $\epsilon = 0.1$, Figure 3.4b shows three regimes: (i) $1 \leq \frac{n}{k} < 1.04$ where the coding rate is too low to recover a meaningful number of packets, but extra coded packets contribute to the aggregate rate, (ii) $1.04 \leq \frac{n}{k} < 1.18$ where the retransmission rate decreases rapidly to 0, compensating for the added coded packets, and (iii) $1.18 \leq \frac{n}{k} < 1.4$ where no retransmissions remain, and increasing the coding rate only loads up the lossy link. A similar behavior can be observed in both scenarios and across our experiments.

The same behavior can be observed in both scenarios.

Figure 3.5 shows similar results. Given the small number of hops in scenario 1, the relative impact of adding packets on the lossy link is more important than for scenario 2. This can be seen in the first and last regimes where the aggregate packet rate increases with the coding



(a) Scenario 1, Theoretical

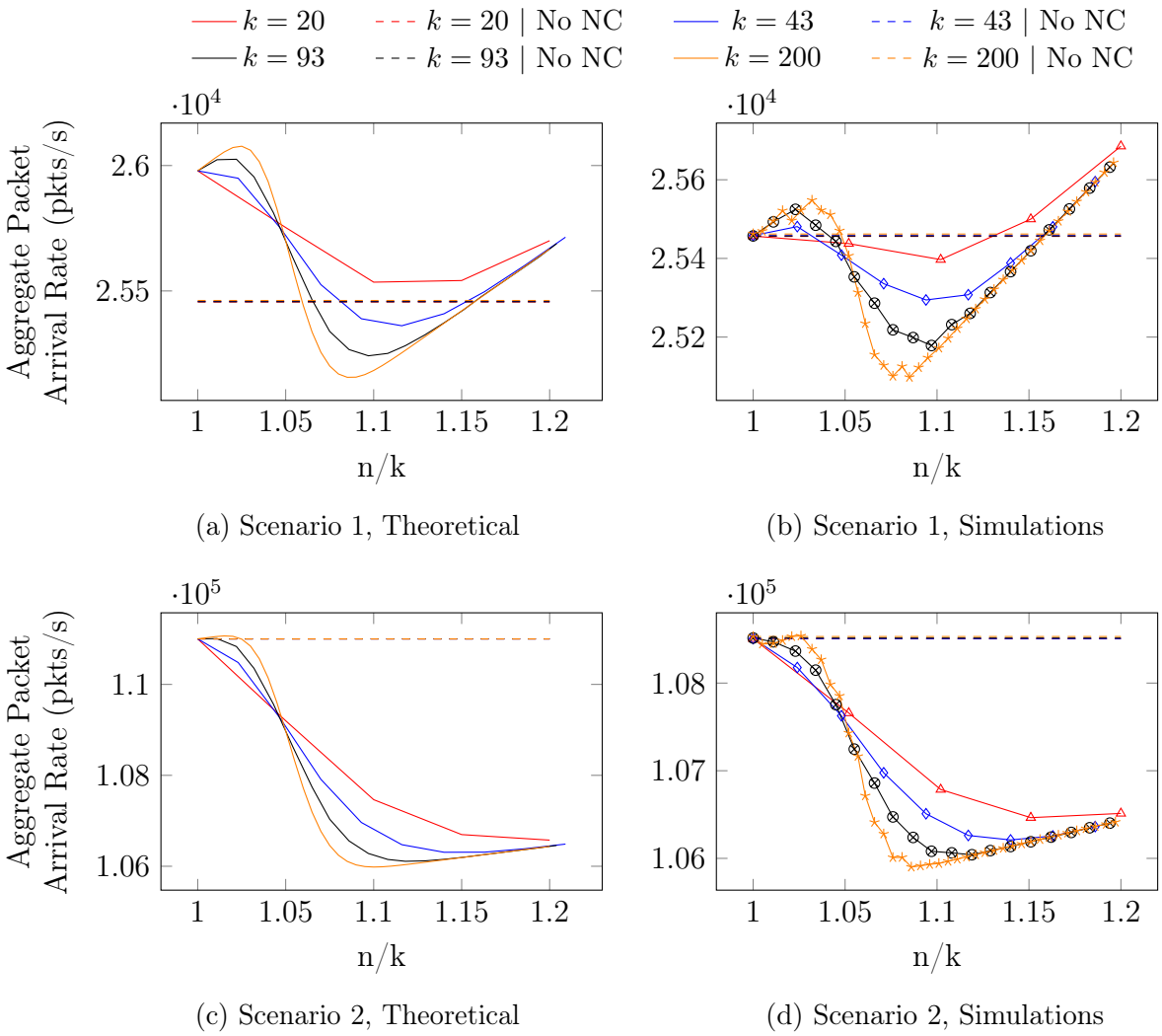
(b) Scenario 1, Simulations

(c) Scenario 2, Theoretical

(d) Scenario 2, Simulations

Figure 3.4: Aggregate packet rate for different erasure probabilities ϵ of the lossy link.

rate. Increasing k has two effects: first, since n and k are integers, the granularity of choice for coding rates is finer when k is large. If k is too small, then the optimal coding rate might not be achievable. Second, our results show that even at a coding rate that can be achieved for all values of k presented here, larger values of k further reduce the aggregate packet rates. As mentioned, our simulations show the same trend as predicted by our analytical model: in practical deployments, for a given ϵ, the optimal coding rate can be computed using equation 3.6.



(a) Scenario 1, Theoretical

(b) Scenario 1, Simulations

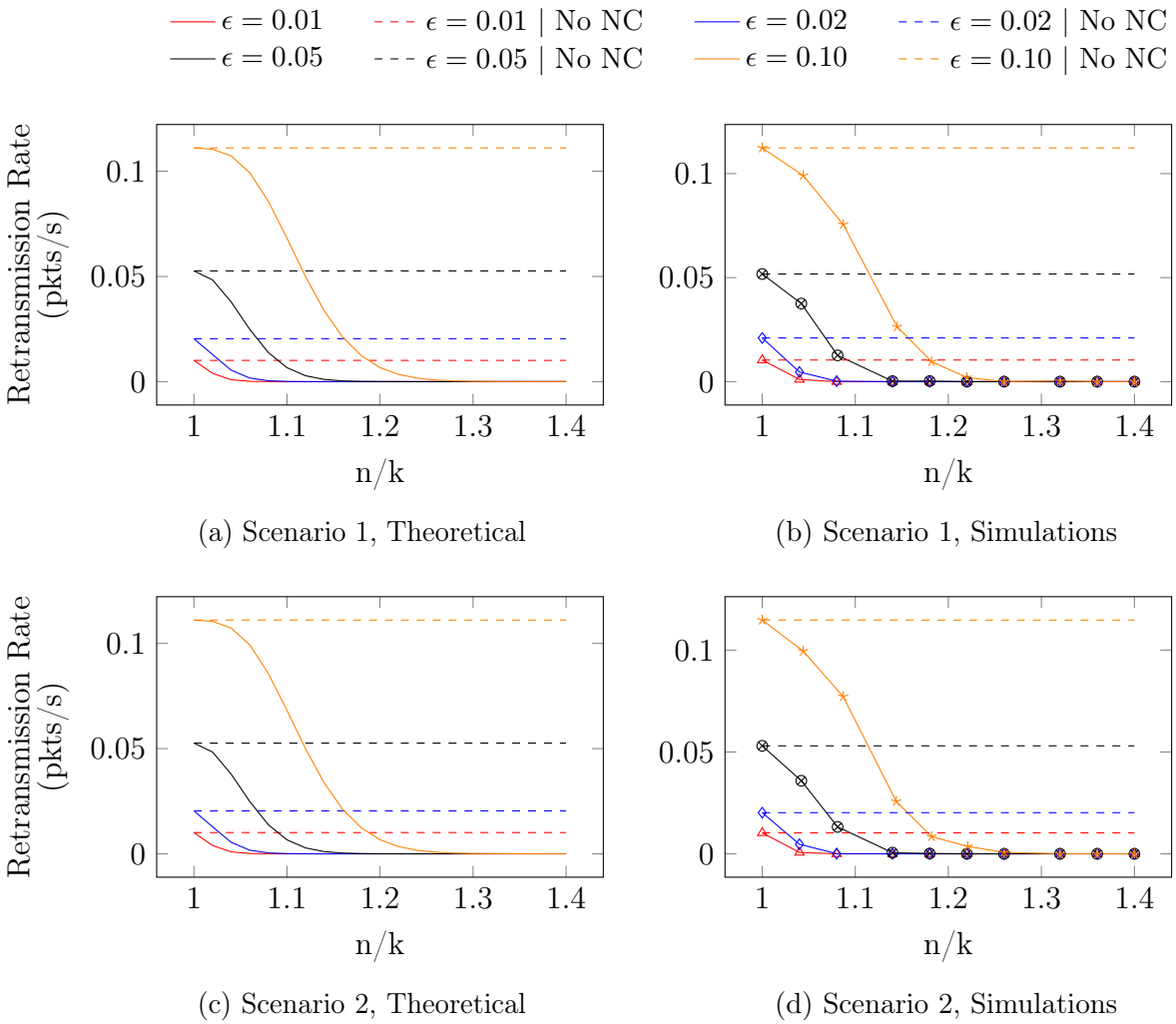
(c) Scenario 2, Theoretical

(d) Scenario 2, Simulations

Figure 3.5: Aggregate packet rate for different values of k.

3.4.3 Retransmission Rate

To evaluate the impact of a reduction in retransmission probability in LINC, Figures 3.6 and 3.7 plot the effects of ϵ and k on retransmission probabilities. In Figures 3.6a and 3.6c, we use equation 3.1 to compute the expected rate of retransmissions and verify in Figures 3.6b and 3.6d that our simulations match the derivation. Importantly, LINC eliminates all unnecessary retransmissions by driving the retransmission rate to zero over the entire path, at the cost of a small added overhead on the single lossy link, for all values of ϵ. In real-world implementations, service providers should use our theoretical model to determine the optimal



(a) Scenario 1, Theoretical

(b) Scenario 1, Simulations

(c) Scenario 2, Theoretical

(d) Scenario 2, Simulations

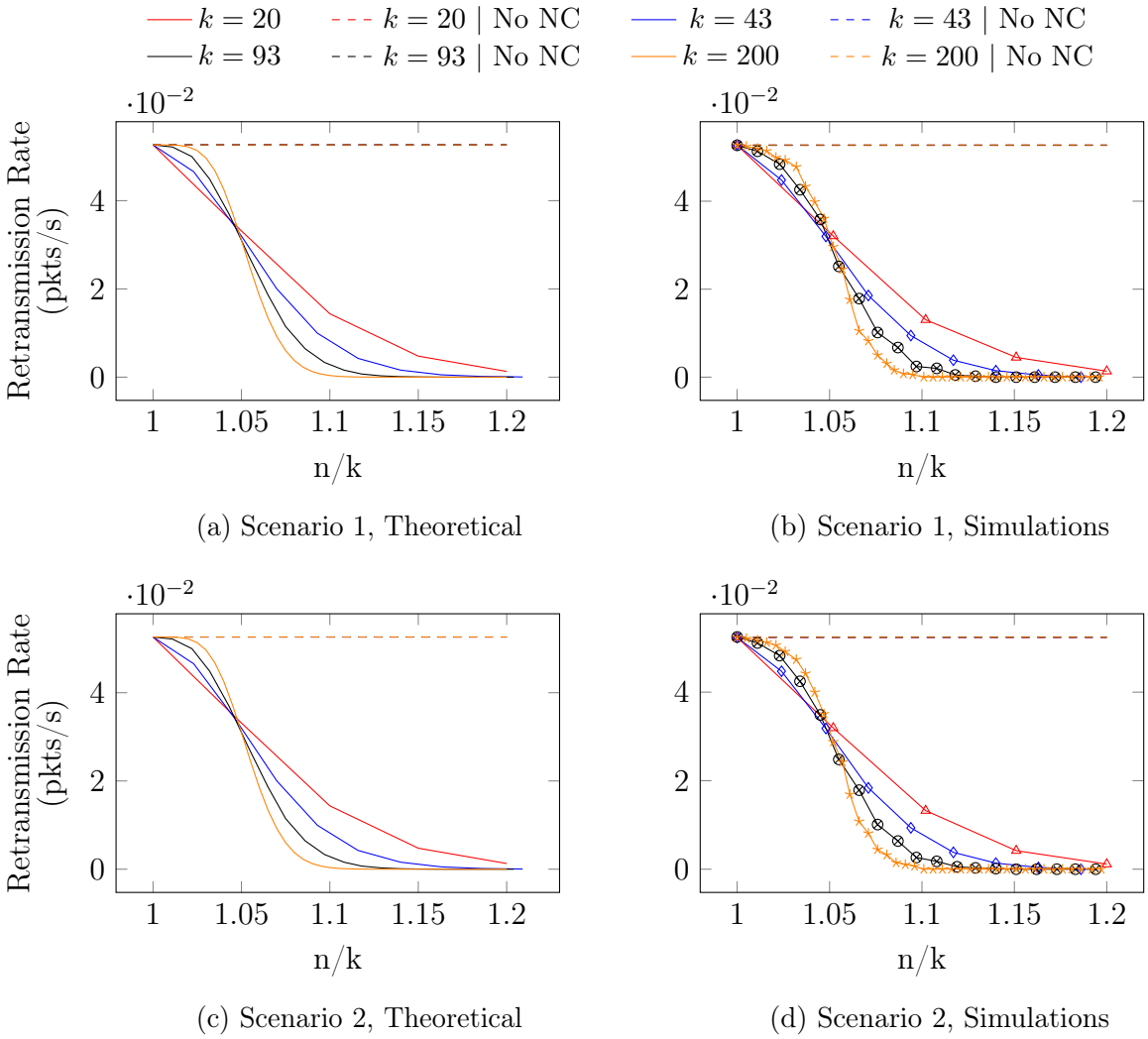
Figure 3.6: Rate of retransmissions for different erasure probabilities ϵ of the lossy link

coding rate in their setup.

Figure 3.7 studies the impact of the value of k on the rate of retransmissions. We observe the same trend as in Figure 3.5: the higher k, the closer the retransmission rate is to a step function. Again, we observe the same trends in simulation as predicted by our analysis.

3.4.4 E2E Delay

Figure 3.8 shows the mean E2E delay experienced by all packets in our simulations. LINC reduces the delay by up to 18 %, by reducing the amount of retransmissions required. For both scenarios, the average delay is primarily related to the probability of a packet being



(a) Scenario 1, Theoretical

(b) Scenario 1, Simulations

(c) Scenario 2, Theoretical

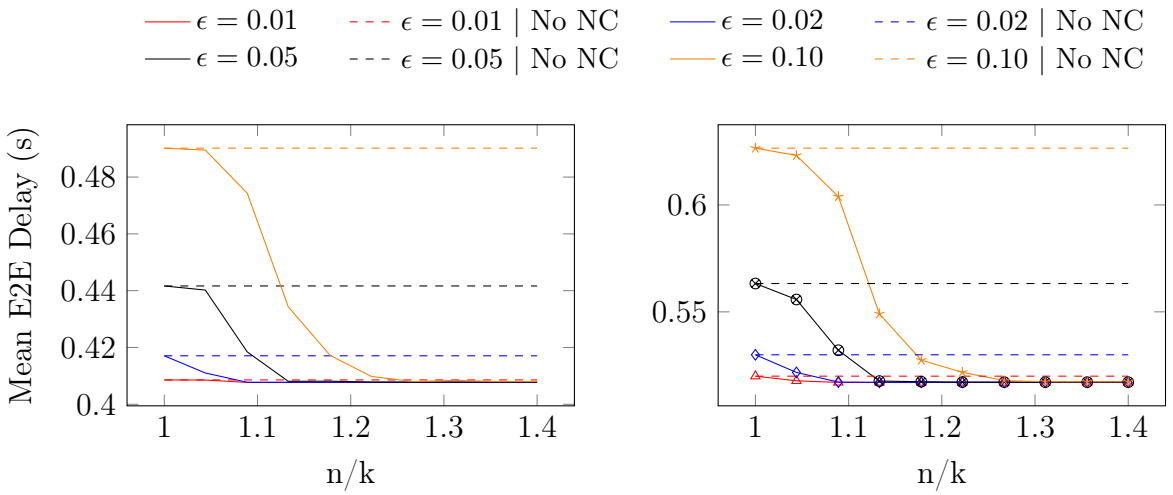
(d) Scenario 2, Simulations

Figure 3.7: Rate of retransmissions for different values of k.

retransmitted, since a retransmission yields a delay of at least 1.5 RTT, compared to 0.5 RTT when a packet is successfully transmitted or recovered on the lossy link. As we increase the coding rate and subsequently decrease the rate of retransmissions, the E2E delay approaches 0.5 RTT.

3.5 Summary and Future Work

In this chapter, we present LINC, an in-network system to tame the impact of environmental packet loss events in hybrid wireless-fiber backbone networks. We provide a detailed analysis



(a) Scenario 1, Mean

(b) Scenario 2, Mean

Figure 3.8: E2E packet delivery delay.

of LINC's goodput gains, providing insights on choosing optimal parameters in practical settings. Our simulations show both the validity of our analysis and that LINC significantly reduces the rate of E2E retransmissions, yielding higher goodput in the network and lower delay for flows subject to packet loss events. Future work will include more sophisticated coding strategies, as well as a programmable switch implementation.

Chapter 4

Sirona: Memory Efficient Reactive Subflow Spraying for Datacenter Load Balancing

Modern datacenter networks employ topologies with rich path diversity between host pairs to improve fault tolerance and enable cost-efficient scaling. However, effectively utilizing this path diversity through load balancing remains a critical challenge. The standard approach, Equal Cost Multi-Path (ECMP), uses flow-level hashing that can lead to persistent congestion due to hash collisions. In particular, ECMP suffers in scenarios with changing path states or varying network conditions, and in heterogeneous clusters presenting asymmetric paths.

In [2], the author introduces Reactive Subflow Spraying (RSS), which combines the benefits of multipath support and reactive routing to achieve near-optimal load balancing. RSS splits flows into subflows that are independently routed. Regularly, RSS uses one-way delay measurements to reroute the worst-performing subflow to a random path. This approach demonstrated flow completion times (FCT) within $5 - 10$ % of optimal across various scenarios.

However, RSS faces significant deployment challenges due to its memory requirements.

The need to maintain state for each subflow can strain memory resources, particularly in high-throughput environments with many concurrent flows. This limitation becomes especially acute when implementing RSS in hardware like SmartNICs, where memory is a precious resource. In this chapter, we introduce Sirona, a memory-efficient version of RSS achieving similar performance with 3× lower memory footprint, making it practical for hardware implementation in modern datacenter environments.

The work in this chapter was done in collaboration with Abdul Kabbani and Ahmad Ghalayini.

4.1 Introduction

Streaming services, social media infrastructure, and the rise of Artificial Intelligence (AI) applications have driven the development of more and more sophisticated datacenters. The networks powering these datacenters are built to provide fault tolerance and cost-efficiency, by offering many different paths to servers wishing to communicate data. Most modern networks use some version of a Fat Tree Topology [50], with flows choosing their path using ECMP [118]. In ECMP, switches in the network hash flow-level information (*e.g.* source and destination IP address, source and destination port, transport protocol) to pick the output port for every packet in the flow. This hash is consistent across the lifetime of the flow, which presents significant benefits for in-order delivery, but also results in persistent congestion when hash collision occurs. Numerous studies have shown that ECMP's oblivious nature results in suboptimal load balancing and can create congestion hotspots [119–133].

The research community has proposed several alternatives to address ECMP's limitations. For example, Random Packet Spraying (RPS) [123] takes a lower granularity approach by distributing individual packets across all available paths, which can significantly improve load balancing in symmetric network conditions. However, RPS's effectiveness deteriorates markedly in asymmetric scenarios, for example in heterogeneous deployments or when link

failures create capacity disparities between different paths. On the other hand, Flowbender [126] reactively reroutes flows experiencing high ECN marking rates and properly handles asymmetries in the network topology. However, since it uses flow-level routing, Flowbender still struggles to achieve perfect load balancing.

At a high level, Reactive Subflow Spraying (RSS) [2] combines the reactivity of Flowbender and some of the finer load balancing capabilities of RPS. Instead of randomly spraying each packet on different paths, RSS takes inspiration from Multi-Path TCP (MPTCP) [122] and introduces subflows as a level of granularity between packet-level (as in RPS) and flow-level (as in ECMP).

RSS faces a significant hurdle in practical deployment: its memory requirements. The need to maintain an independent state for each subflow imposes substantial memory overhead, particularly in environments with high numbers of concurrent flows. This limitation becomes particularly problematic when considering implementation in hardware accelerators such as SmartNICs [134], where memory resources are both limited and expensive. As datacenter networks continue to scale and handle increasing numbers of concurrent flows, this memory overhead threatens to make RSS impractical for widespread deployment.

In this chapter, we build on [2] and present the following extensions of RSS:

- We explore the importance of the choice of congestion signal for the performance of RSS.
- We provide insight into the choice of parameters for RSS.
- We present our study of various flavors of RSS.
- We introduce Sirona, a memory-efficient version of RSS achieving similar performance with 3× lower memory footprint, making it practical for hardware implementation in modern datacenter environments.

Through extensive simulations, we show reductions in maximum FCT of up to 89 % over traditional load balancing schemes, we demonstrate that Sirona performs just as well as RSS

(within less than 4%), and provide a sensitivity analysis of various parameters of RSS not yet explored.

4.2 Background

We reproduce here for clarity arguments motivating RSS exposed in [2], and discuss the limitations of the initial design.

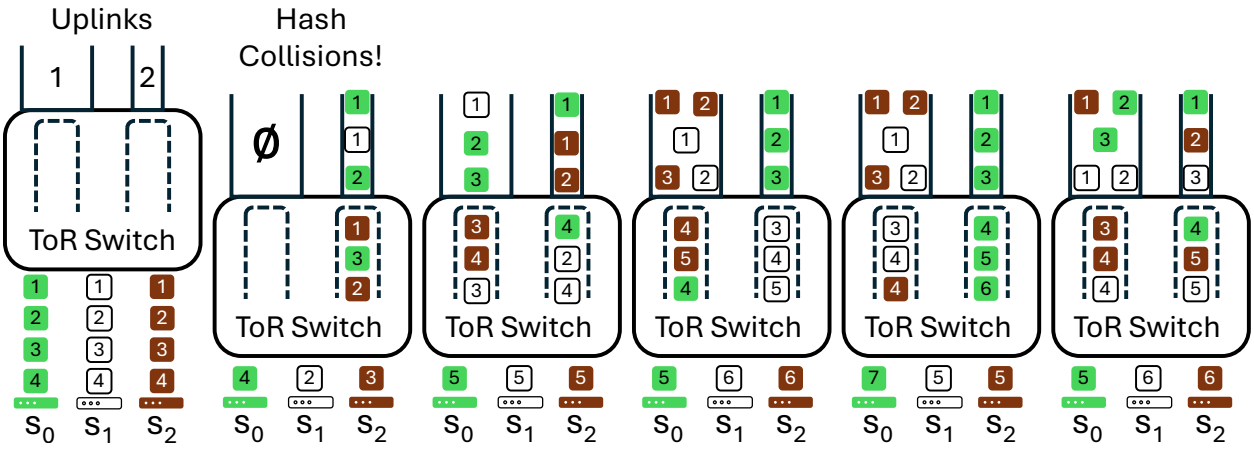
4.2.1 Background

As mentioned in section 4.1, modern datacenter networks offer many redundant possible paths for network traffic. Load balancing can be seen as a balanced allocation problem: n flows need to be spread over p paths as close to uniformly as possible. On a conceptual level, this problem of placing balls into bins has been extensively studied (see for example [135–138]): as the number of balls–or flows in our case–grows, the distribution of the number of balls per bin tends towards a uniform distribution. As shown in [136], the maximum load for $n > p \log p$ is $\frac{n}{p}\left[1 + \Theta\left(\sqrt{\frac{p \log p}{n}}\right)\right]$, hence for $n \to \infty$, the load of the busiest bin and therefore of all the bins tends to $\frac{n}{p}$. Outside of this asymptotic regime, severe imbalances can appear. For example, when the number of flows equals the number p of available paths, the probability of at least two flows colliding (and therefore at least one path being idle) is $\frac{n-1}{n}$.

We show in Figure 4.1a a sample Top of Rack (ToR) switch with 5 ports and 3 servers to illustrate the main differences between the classes of load-balancers described in this section. Note that the diagram presents some asymmetry, with the right uplink having a lower capacity than the left one.

ECMP

ECMP is widely used across modern production datacenters. In ECMP, when a switch receives a packet, it inspects its header, and hashes flow level information (collectively denoted by the



(a) ToR Switch (b) ECMP (c) RPS (d) MPTCP (e) Flowbender (f) RSS

Figure 4.1: (a) Sample ToR switch with 5 ports. (b-f) Example routing decisions for different load balancing approaches.

packet's *entropy*) to randomly decide among valid candidates which output port to send the packet on. This decision is uniformly random across different flows, but consistent within a flow. This has the benefit of ensuring in-order packet delivery which traditional transport protocols like TCP rely on.

This granularity is the first limitation of ECMP: suppose a sender is sending five different flows to a single receiver, in a cluster offering four different paths. In the best-case scenario, one path will host two flows, and the remaining three paths will each host a single flow. Hence, the two flows sharing a path will have significantly lower quality of service–and higher Flow Completion Time (FCT)– than the other three flows.

The second limitation of ECMP comes from its randomness. Switches make a one-shot routing decision that persists throughout the lifetime of the flow. In the worst-case scenario of our example, all 3 flows could be hashed to the same path, resulting in 3× higher FCT, and three completely idle paths. Even in cases where demand and offer match, as in a case with four flows and four distinct paths, the randomness of ECMP could lead to an unbalanced allocation. Since ECMP cannot react to network conditions, no rebalancing would occur in either of these examples. The probability of this worst-case scenario, and generally of very poor load balancing scenarios happening is inversely proportional to the number of flows

sharing the system.

Figure 4.1b shows one such poor load balancing scenario, with all three servers s_0, s_1, and s_2 sharing the low bandwidth–link. While all three servers are reducing their sending rates to accommodate sharing their allocated paths, the first uplink sits idle.

Random Packet Spraying (RPS) [123]

On the other side of the granularity spectrum, RPS has senders change part of the flow's signature in the entropy for *every* packet. The same hashing process happens at the switches, but this time producing a new hash for each packet. Therefore, a new random path is chosen for every packet, effectively randomly spraying packets across all available paths. As we have discussed, increasing the number of balls (from flows to individual packets) enables close to uniform load balancing: on average, every path will receive the same number of packets. While series of packets could hash to the same path, these events have a probability vanishing with their duration and importance.

The main drawback of RPS is highlighted by asymmetric conditions: suppose that due to hardware differences, some links $\{l_i\}$ have slower bandwidth than others. The sender is unaware of this imbalance and therefore continues to randomly assign an entropy for each packet. Since the flow of packets going through any of the $\{l_i\}$ will have a slower rate, the congestion control algorithm will adjust the rate for all paths, leading to underutilization of all the other links.

Figure 4.1c shows this effect: all packets from all flows are sprayed equally across both paths. Since the right link has lower capacity, a queue builds up, eventually leading to packet drops, ECN markings, or some other congestion signal. The sending rate is therefore adjusted to the capacity of the worst path, leading to an underutilization of the first uplink.

MPTCP [122]

Finding a balance between per-flow and per-packet granularity has been explored in some way in MPTCP. In that proposal, TCP flows are split into subflows each using a different network path, yielding better resource utilization, increased redundancy, and failure resiliency. Each subflow maintains its own congestion state, including its own congestion window (and therefore its sending rate).

However, MPTCP does not allow for dynamic rerouting of subflows: the choice of paths is done at the start of the connection and remains for its duration. Subflows can react to varying network conditions by changing their rates but are constrained to a fixed routing path. On top of this, the inherent complexity of keeping per-subflow congestion state poses challenges for hardware-deployments [126].

Flowbender [126]

We discussed in the previous three sections the differences between ECMP, MPTCP, and RPS in terms of load balancing granularity and the quality of the one-shot balancing decision. Flowbender looks at the orthogonal problem of adjusting path assignments dynamically as network conditions change. More specifically, Flowbender adopts the same granularity as ECMP but uses Explicit Congestion Notifications (ECN) [139] markings as signals for collision-based congestion. ECN-enabled switches set the ECN flag in packets' headers when congestion is likely to happen (in practice when the queue increases past some threshold). When a Flowbender sender detects a fraction of ECN marked acknowledgments higher than some predefined threshold for some time, it changes the entropy of the flow, resulting in a new path being chosen for the flow with probability $\frac{p-1}{p}$. Flowbender does not need information about the topology of the network: changing the entropy of a flow has a high probability of changing the path that the flow will take.

The main idea behind Flowbender can be seen as a series of dice rolls, where only the lowest-scoring dice are rerolled at each round. Naturally, in this metaphor, the total score

for all the dice increases with time. Similarly, for long-running flows, Flowbender converges towards a better allocation of flows, with congested paths from hash collisions rerouted until a good path has been found.

4.2.2 Reactive Subflow Spraying (RSS) [2]

Load balancing schemes can be partially characterized by the routable unit on which they make decisions: flows for ECMP, and packets for RPS. RSS sits in the middle of this spectrum and includes reactivity to network conditions with the same approach as Flowbender. Packets are grouped in *subflows* in a round-robin fashion at the sender, with the number of subflows being a tunable parameter σ set by the sender. This yields a resulting rate for every subflow equal to $1/\sigma$ of the rate of the flow. Instead of handling load balancing by adjusting the rates of subflow, we adjust the number of subflows per path to approach proper load balancing.

The number of subflows acts as a dial on the granularity scale: ECMP corresponds to having a single subflow, and RPS corresponds to having an infinite number of subflows. In practice, for hardware deployability reasons, we choose to limit σ to 16, but study its impact in section 4.4.4.

As it receives acknowledgments, an RSS sender records the RTT values experienced by each subflow (*i.e.* RSS uses RTT as a congestion signal). Periodically, RSS compares the average RTT for each subflow over that time period and reroutes the subflow by changing its entropy.

RSS only assumes that switches in the network support ECMP: all the functionality is implemented on the sender, and packets from a flow look to switches like they belong to different flows. Conversely, RSS needs no information about the network topology, but instead naturally probes paths by adjusting the entropy of individual packets. Compared to Flowbender, RSS explores network paths $\sigma\times$ faster.

Like other non-flow-based load balancers, RSS assumes that the transport protocol handles out-of-order packet delivery. Indeed, a packet that is stuck for some time in a switch buffer

can be overtaken by its successor follows a different path. Such network stacks have been designed recently (*e.g.* SRD [140]). RSS's design has the benefit of being much simpler to implement than other multipathing approaches like MPTCP: all subflows remain part of the same connection with the same parameters. The round-robin assignment of packets to flow has other benefits, in particular in terms of loss recovery, that are out of the scope of this work.

4.3 Sirona: Extending RSS

4.3.1 Why RSS works

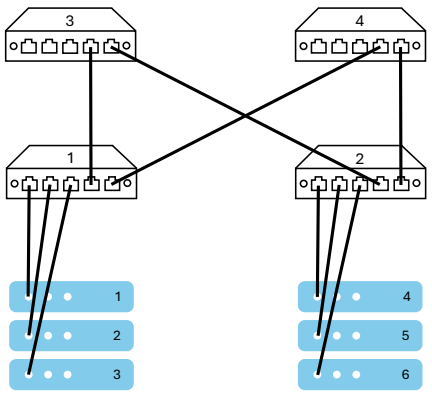


Figure 4.2: Oversubscribed topology with 8 servers, 2 layers, and 4 servers per Top of Rack (ToR) switch.

As we detailed in section 4.2, load balancing problems can be modeled as a balanced allocations problem. Consider the 6-node cluster shown in Figure 4.2, and a traffic matrix where the three servers on the left $(1-3)$ send traffic to the three servers on the right $(4-6)$. Each sender generates σ subflows, hence a total of 3σ subflows handled by switch 1. Since both paths (going through switch 3 or switch 4) have the same capacity, the optimal

allocation consists of allocating 1.5σ subflows to each path. In cases where σ is odd and 1.5σ is not an integer, there are two optimal allocations that RSS will oscillate between.

Therefore, we can model the system with two bins and 3σ balls. Initially, all balls are placed randomly. Then, periodically, 3 balls (one per sender) are removed from the fullest bin and replaced in random bins. In this simple example with only two bins, the situation is guaranteed to not degrade until oscillation around the optimal allocations is reached. At every step until then, either balls are placed in the same bin they were removed from, and the situation stays the same, or some of the replaced balls are put in a different bin, and the situation improves. The probability of the first case being low (in this particular case, 1/8), the system will stabilize around the optimal allocation. When an optimal solution is reached, RSS keeps rerouting flows, therefore a worse allocation could be visited temporarily.

RSS has two components: dividing flows into subflows and reactively rerouting subflows. RSS practically increases the number of balls to be placed in bins, hence it is naturally likely to have a better allocation than ECMP. Its reactivity acts as a second layer that improves allocations over time.

Large settings make the analysis less direct: for example, topologies with three tiers should be modeled as interdependent allocation problems since the links between tier $1-2$ and tiers $2-3$ do not see the same subflows. Furthermore, since each sender reroutes one flow per update interval, when many senders are present, only one ball from the fullest bin is guaranteed to be moved. In any case, at each iteration, balls are taken from some of the busier bins and are replaced randomly across all bins, therefore likely to improve the allocation.

4.3.2 RSS Memory Requirements

RSS, as described in [2] needs the following to function, for each flow:

- A vector E of size σ to record the entropy of each subflow. To make sure each subflow keeps the same path until it gets rerouted, the sender must keep a record of the entropy

value for each subflow.

- A vector Σ of size σ to record the running sum of RTTs for each subflow.
- A vector N of size σ to record the number of packets that were acknowledged for each subflow. To normalize the sum of the received RTT values properly, the sender must keep track of how many packets were ACKed for each subflow.

Other individual registers are needed, for example, to keep track of time and periodically reroute subflows, but the main memory footprint comes from the above. This memory footprint comes on top of the state information needed by the congestion control algorithm. As described in section 4.3.1, higher subflow counts should help with reaching an optimal load balancing state. However, Network Interface Cards' (NIC) hardware only has so much memory. We therefore explore modifications to RSS details to attempt to reduce the memory footprint of RSS.

4.3.3 RSS Flavors

The description of RSS in [2] considers two parameters: the number of subflows, and the duration of the period between two rerouting events. In practice, many other choices could impact the final performance and the memory footprint of the system, such as the choice of the congestion signal, or the number of rerouted flows per period. We present in this section different flavors of RSS that we experimented with, along with a memory analysis.

Congestion Signals

[2] computes the mean Round Trip Time (RTT) to determine the worst performing subflow. The main signal that RSS needs to function is that of congestion: it needs to figure out which subflow sits on the busiest path. Therefore, only recording the maximum value of the RTT over some time period rather than computing the mean should give access to similar information. Focusing on the maximum value removes the need for vector N, since no

normalization operation is necessary, and requires a vector M instead of Σ of size σ to record the maximum RTT reported by each subflow. Since many commercial switches now support ECN, and since many state-of-the-art datacenter congestion control protocols rely on ECN (*e.g* DCTCP [121]), we also consider ECN percentage as an alternative to mean RTT. Using ECN does not change the memory footprint of RSS, since computing a percentage requires knowing the total number of packets.

Should subflows with no feedback be rerouted?

For every congestion signal, we must figure out whether to reroute in priority subflows for which no feedback was received can be asked. In practice there can be two reasons for this: either (i) a path is severely congested, to the point where packets are dropped, or acknowledgements are delayed beyond an update interval, or (ii) the congestion window of the flow is smaller than the number of subflows, and some subflows did not get to send any packets during this update period. In the first case, these subflows should be rerouted. In the second case, rerouting them would lead to instability. We verify empirically in section 4.4.5 that in cases of interest, the first situation is more represented. We refer to the scheme rerouting only subflows for which feedback was received as Feedback-Dependant RSS (FDRSS). To account for both cases, we could make the number of subflows to be dynamic, so that all subflows are used within an update interval. We leave that extension for future work.

In any case, if N is not required by the congestion signal, we store information about whether a subflow received path feedback using a one-hot vector B of σ bits.

Number of subflows rerouted per period

Suppose in the example of Figure 4.2 that all subflows for all senders choose the same path. Before the optimal allocation is reached, half of the subflows (1.5σ) need to be rerouted, and each update period sees 3 subflows rerouted. Therefore, $\sigma/2$ update periods are required, at minimum, before the equilibrium is reached. In the transitory period where subflows are not

yet load-balanced properly, the system will have poor performance. The duration of that transitory period has an impact on the average performance of the system. When RSS has to deal with mouse flows, this might mean that the transitory period lasts for more than the lifetime of the flow.

On the other hand, since RSS does not stop rerouting subflows even when a good load balancing solution has been reached, the tradeoff is the following: rerouting more subflows per update interval yields a short transitory period, but larger oscillations in the stable regime.

Rerouting multiple subflows per update period does not increase the memory footprint for the schemes described earlier.

Frozen RSS

The initial allocation of subflows is random. The reactive aspect of RSS moves the assignment of subflows to paths towards a uniform distribution. Once an optimal distribution is reached, RSS still keeps on rerouting flows. Therefore, we introduce Frozen RSS, which measures at each update interval how far the worst-performing subflow is from the best one and refrains from rerouting any flow in cases where the two values are within some margin of each other. The idea behind this scheme is that in the steady state regime, if no flows arrive or finish, rerouting flows will push the system away from the optimal solution, and keep it oscillating around an optimal allocation. If the optimum can be detected, then no rerouting is necessary.

The challenge with this scheme is to pick the correct threshold: a high threshold ensures stability, but risks stabilizing far away from the optimal allocation.

Skipping some update periods does not increase the memory footprint for the schemes described earlier.

Random-Time RSS

RSS uses a time period to regularly update the routing assignment of subflows. To test if this periodicity is necessary, we introduce Random-Time RSS. Instead of periodically

checking for congestion signals, Random-Time RSScan reroute flows on any packets. For every ECN marked acknowledgment, the sender reroutes its corresponding subflow with a small probability. We add a cooldown period after a subflow is rerouted where no ECN mark trigger reroutes so that the intervention has time to take effect.

Random-Time RSSdoes not need N, Σ, or B. It therefore only requires keeping track of the assignment between subflows and entropies.

MPTCP-inspired (QPRSS)

Through hardware implementation discussions with collaborators, it appeared that one simple way to implement RSS on available hardware was to allocate one RDMA-like Queue Pair per subflow. For practical reasons, this leads us to consider a scheme where each subflow keeps its own congestion information. The subflows update their congestion window as if they were flows. When a packet is about to be sent, the number of in-flight bytes for the next subflow is compared with its congestion window. If the packet cannot be sent on this subflow, the same attempt is made for the next subflow, until a subflow that can accommodate the packet is found. If none of the subflows have credit to send the packet, it is delayed until the initial subflow can accommodate it.

On a high level, we can expect the reactivity of RSS to play a synergetic role with congestion control dynamics, both mechanisms having complementary reactions to congested paths. Practically, this means that both share the load balancing burden and that similar results to RSS should be observed with fewer subflows. From a memory standpoint, this adds significant demand since congestion information needs to be replicated across subflows. This scheme breaks a desirable property of RSS for loss recovery. Since RSS packets are sent in a round-robin fashion, a receiver knows that all packets with sequence number $n \equiv i \mod \sigma$ belong to subflow i. Therefore, if a receiver receives packets n and $n + 2\sigma$, it can assume that packet $n + \sigma$ was lost. The round-robin assumption does not hold for QPRSS, hence loss detection at the receiver is significantly more challenging. However, loss recovery is outside

the scope of this work, thus we leave the exploration of methods to enable loss recovery for QPRSS for future work.

Sirona

Finally, we noticed that the most recent congestion signals should be weighted more than the signals from the start of the update period. Indeed, after rerouting a subflow, a packet acknowledgment might come in at the beginning of the next period, and would not give information about the impact of rerouting that flow. Therefore, we introduce two additional schemes. First EWMA, which determines the worst performing subflow by computing an exponential window moving average of the RTT for each subflow. EWMA does not need to keep track of the number of packets received per subflow (N). Second, we propose Sirona, which only looks at the worst measured RTT over a fraction of the update interval. Sirona still relies on update periods but ignores all congestion signals or metrics coming in the first $100 - f$ % of the update interval. In the last f %, Sirona keeps two registers: one for the maximum RTT value encountered across all subflows, and one for the id of the subflow that experienced this RTT.

4.4 Evaluations

4.4.1 Methodology

We run extensive simulations to provide insight into RSS and Sirona, and demonstrate up to 89 %, 41 %, and 42 % gains in max FCT over ECMP, Flowbender, and RPS respectively.

Simulation Environment

We use an extension of the htsim simulator [141], to simulate 1,024 nodes, 2-tiered fat-tree topologies with no over-subscription. We set the MTU to 2KiB and each link's latency to 1.5 μs (*i.e.* a minimum RTT of 12 μs for traffic traversing both tiers). We compare

two network settings: a symmetric case, with all links having 800Gbps capacity, and an asymmetric case, with 8 out of the 32 aggregate switches having all of their links reduced to 400Gbps. The congestion control algorithm used across experiments is DCTCP [121]. To separate the study of Sirona from loss recovery problems, experiments ran here assume that switches trim packets [141] and only forward headers when their queues are full and when they would have otherwise dropped packets. This feature is part of the specifications of leading industry working groups [142]. The initial congestion window for all flows is set to 50 MTUs, or about 2 bandwidth-delay product.

For all experiments except sections 4.4.4 and 4.4.9, we set the number of subflows to 16, and the update interval to 32 μs, or about 4 minRTT. We add a 25% jitter to the update interval for every sender, to avoid rerouting synchronizations for experiments using some version of RSS.

We report mean and maximum FCT as the main metrics for load balancing performance. Since poor load balancing decisions translate into some paths being over-utilized and some paths staying idle, FCT is directly proportional to the quality of the load balancing. In particular, when all flows are of the same size, the maximum FCT will be experienced by the flows competing with most other flows on their path.

Workload

For all experiments except in section 4.4.8 and 4.4.9, we consider the following traffic pattern: we create pairs of senders and receivers where senders are in the first half of the nodes $(\#0-511)$ and receivers are in the second half of the nodes$(\#512-1023)$. The exact pairings are chosen at random, such that each receiver receives traffic from a single sender, and each sender sends traffic to a single receiver. This simulates point-to-point connections that maximize the stress on the load balancer.

Schemes

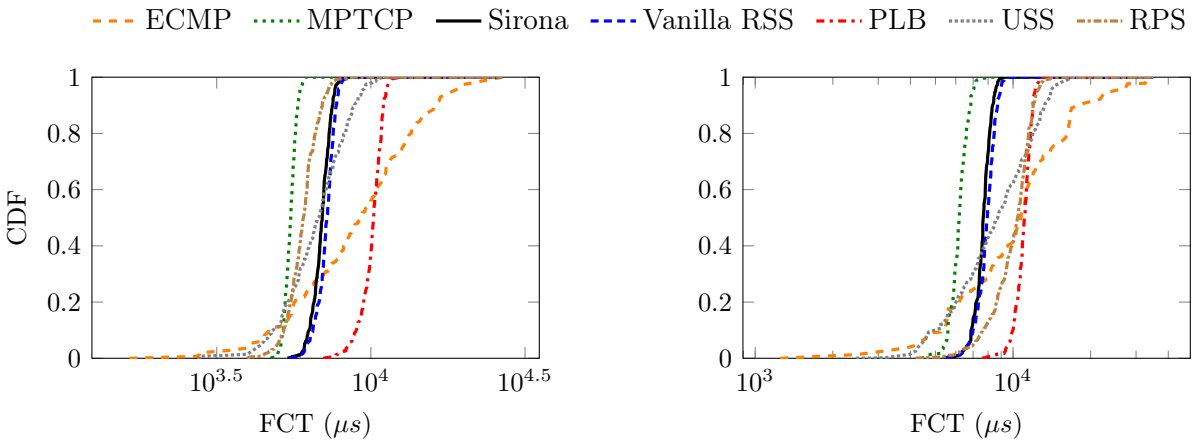
We consider three classes of schemes that we compare Sirona to: baselines, RSS flavors, and congestion metrics.

Baselines:

- ECMP (see 4.2.1): hashes flow-level information to pick a single path for all packets in a flow, and maintains that path for the lifetime of the flow.
- Flowbender (see 4.2.1): similar to ECMP but proactively reroutes flows that see high ECN marking rates. We choose the same parameters as advised in the paper, *i.e.* an ECN threshold of 0.05 and several consecutive congested periods before choosing to reroute of 1.
- RPS (see 4.2.1): chooses a new random path for every packet, but is oblivious to path heterogeneity or network conditions.
- MPTCP (see 4.1d): opens different connections on each subflow, trading high memory utilization and complexity for better performances.
- Unreactive Subflow Spraying (USS): To differentiate between the reactive and the subflow spraying aspects of RSS, we introduce USS, with which flows are divided into randomly routed subflows, but senders do not change subflows' routing during their lifetime. USS should perform slightly worse than RPS given the coarser granularity. We set the number of subflows to be 16, for a fair comparison with Sirona.
- Vanilla RSS: as described in 4.2.2 and [2].

Flavors: To understand the impact of various parameters on Sirona's performance, we compare the following versions of RSS:

- Random-Time RSS (RTRSS, see 4.3.3): instead of periodically rerouting subflows, randomly choose to do so when receiving congestion signals.



(a) Symmetric

(b) Asymmetric

Figure 4.3: FCT distribution for baseline schemes and Sirona.

- Frozen RSS (see 4.3.3): when the variance across subflows' congestion metric is low enough, skip the update interval.
- QPRSS (see 4.3.3): perform congestion control on a subflow basis.
- FDRSS (see 4.3.3): reroute the worst performing subflow among those for which feedback was received.

Metrics: To determine which subflow is performing the worst and should be rerouted, multiple metrics can be used:

- Vanilla RSS: use the average of RTT values received in acknowledgments over the update period.
- EWMA: Exponential Window Moving Average of the RTT for each subflow.
- Max: the maximum RTT value experienced for a subflow over the entire update interval.
- ECN: the fraction of ECN marked acknowledgments received.

4.4.2 Sirona fairs similarly to RSS

We show in Figure 4.3 the Cumulative Distribution Function (CDF) of the FCT for all the baseline schemes. Figures 4.3a and 4.3b respectively show results for the symmetric and asymmetric cases described in section 4.4.1.

In both cases, ECMP has a high risk of hash collision. In the asymmetric case, it has the added issue of having an equal probability of assigning flows to the 800Gbps and 400Gbps paths. Hence, ECMP yields 50 % and 81 % (respectively 51 % and 89 %) higher average and max FCT for the symmetric setup (respectively the asymmetric setup) than Sirona.

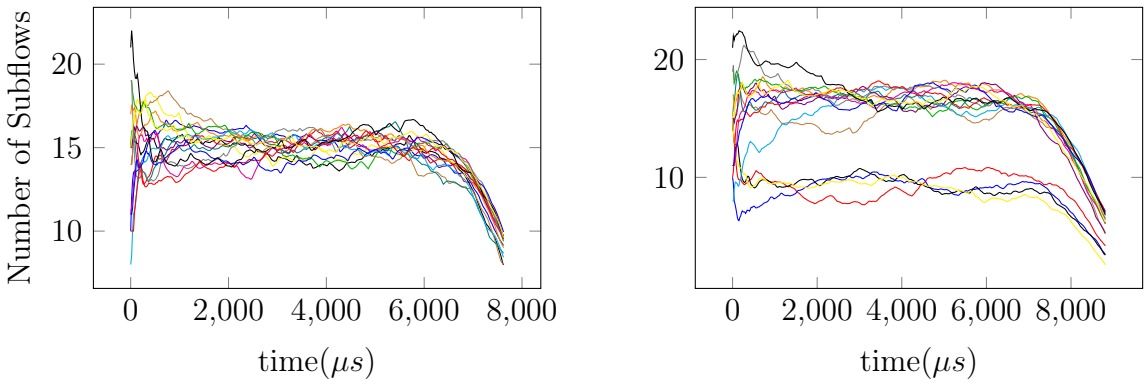
USS has finer granularity than ECMP, but is not reactive and therefore suffers from the same problem as ECMP in the asymmetric setup. Therefore, while it results in only 4 % higher mean and 42 % higher max FCT in the symmetric case than Sirona, these numbers shoot up to 27 % and 67 % respectively in the asymmetric case. The reactive nature of RSS and Sirona is therefore detrimental to performance, in particular in asymmetric setups.

Flowbender is on the same granularity level as ECMP, and therefore can only achieve non-optimal load balancing. Its rerouting capabilities make it suffer less from heterogeneity in the offered paths, with 43% and 42% higher mean and max FCT than Sirona in the symmetric case, compared to 42% higher both mean and max FCT than Sirona.

RPS achieves very high-performance load balancing in symmetric setups, as it has per-packet load balancing granularity. In the symmetric case, it outperforms Sirona in mean FCT slightly (3%) and does only slightly worse in max FCT (21%). In the asymmetric case, it is forced to reduce the sending rate for all paths due to the 400Gbps links and performs 23% and 42% worse than Sirona in mean and max FCT.

As expected, MPTCP makes proper use of subflows and is able to outperform all other schemes. Sirona is within 26% in mean FCT (23%) and 30% in max FCT (16%) in the symmetric case (respectively the asymmetric case).

Finally, Sirona is within 2% in both mean and max FCT of RSS, for both symmetric and



(a) Symmetric

(b) Asymmetric

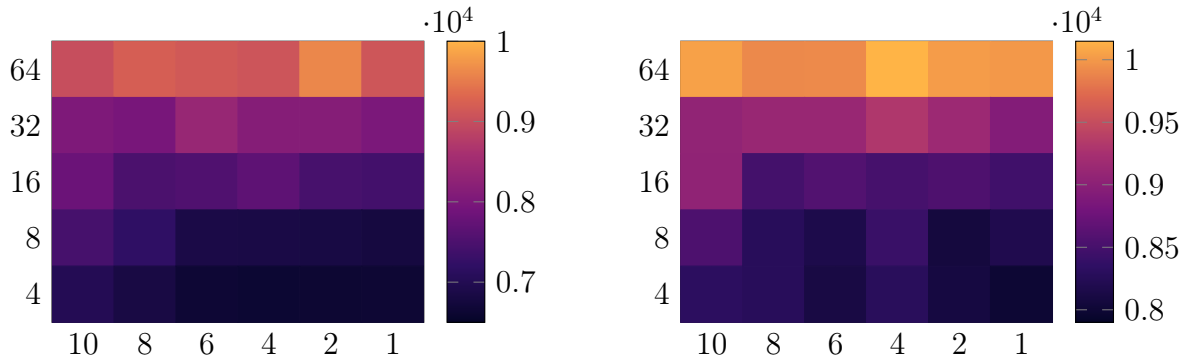
Figure 4.4: Timeseries of the number of subflows using each path. Each solid line corresponds to an exit port of the switch. Black dashed lines represent the optimal number of subflows per path.

asymmetric cases, showing that it yields the same benefits in general while requiring much less memory.

4.4.3 Load Balancing Performance

Figure 4.4 shows a time series of the number of subflows assigned to each path for Sirona. In the same experiment as in Figure 4.3, we record for each packet coming out of one of the ToR switches hosting senders the timestamp at which it departed the switch, the subflow, and the sender to which the packet belongs. We then count in increments of 1.5 update intervals the number of unique (sender ID, subflow ID) coming out of each port of the switch. Results are then averaged using a rolling window.

In the symmetric setup, all paths are similar, hence we expect them to all hold the same number of subflows. Figure 4.4a shows this behavior, with the number of subflows per path oscillating around the expected value represented by the black dashed line. In Figure 4.4b, 8 of the 32 aggregate switches (and therefore 8/32 paths) have a reduced capacity. Sirona successfully adapts and reduces the number of subflows for the 8 slow paths to about half of that of the full capacity paths.



(a) Symmetric (b) Asymmetric

Figure 4.5: Heatmap of the maximum completion time as a function of the number of subflows and the update interval (in units of min RTT).

4.4.4 Sensitivity Analysis

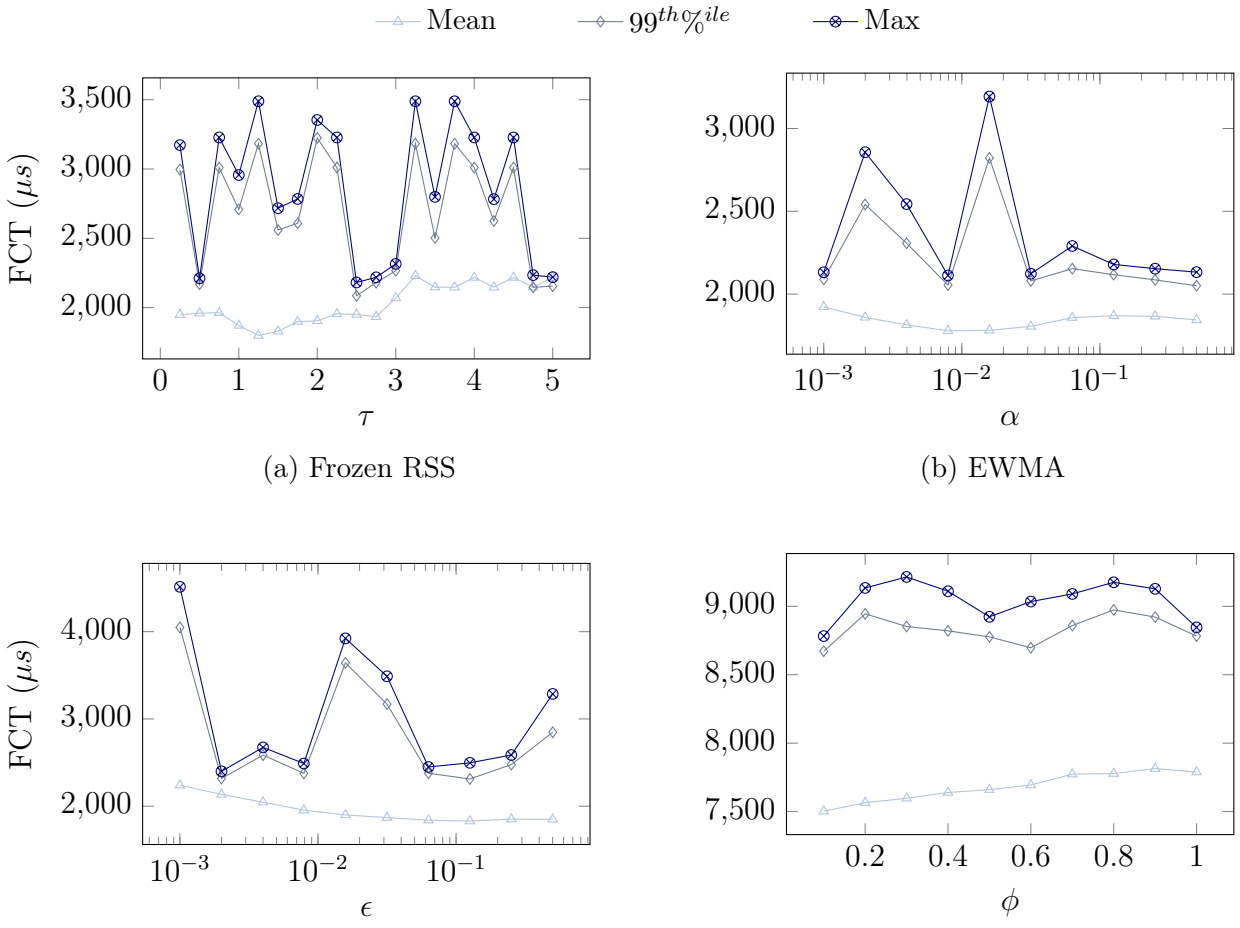
We inspect the sensitivity of some of the different versions of RSS we introduced to the values of their parameters. Since the asymmetric scenario puts more stress on the load balancing capacities, we only show results for that topology in this section.

Vanilla RSS

We show in Figure 4.5 the impact of the original RSS parameters on the maximum FCT. In general, adding more flows helps with reducing the FCT: since the granularity of the load balancing is finer, better load balancing solutions can be attained. Similarly, shorter update intervals help with both making RSS more reactive and reducing the transitory period. However, a combination of very short update intervals and a high number of subflows yields poor results. When the update interval is too short to ensure that all subflows are able to send packets and receive feedback on their current path, some subflows on underutilized paths could be rerouted while subflows on congested paths will never have priority.

Frozen RSS

In Frozen RSS, at the end of each update period, the relative difference between the congestion metric of the worst-performing subflow and the best-performing subflows is compared to a



(a) Frozen RSS

(b) EWMA

(c) Random-Time RSS

(d) Sirona

Figure 4.6: Sensitivity analysis for different flavors of RSS.

threshold τ. We plot in figure 4.6a the impact of this threshold on the mean, 99$^{\text{th}}$ percentile, and max FCT. When the threshold is too low, very few or no updates are skipped, missing the impact of this scheme. When the threshold is too high, the scheme stabilizes far from the optimal load balancing allocation.

While there exists a value of the threshold that minimizes the mean FCT for this setup, the value of the maximum flow completion time varies significantly with no clear pattern. Therefore, it does not seem possible to determine a clear value for the parameter.

EWMA

When a new packet acknowledgement with RTT RTT_{emp} is received, the new RTT estimate RTT_{new} for the corresponding subflow is updated from its previous value RTT_{old} in the following way:

$$RTT_{new} = \alpha RTT_{emp} + (1 - \alpha) RTT_{old}$$

Therefore, the parameter α determines the relative importance of new measurements in the estimation of the RTT.

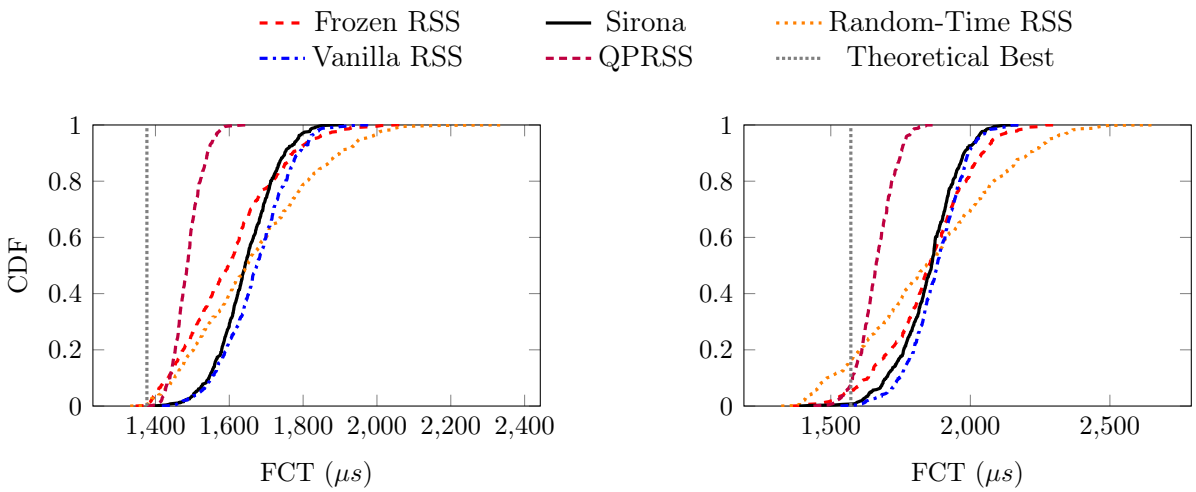
Figure 4.6b shows how increasing the weight of the last RTT measurements improves the performance of RSS with EWMA.

Random-Time RSS

In Random-Time RSS, each packet acknowledgment marked with ECN has a probability ϵ of triggering a rerouting for the subflow it belongs to, as long as the subflow was not already rerouted recently. The value of the probability threshold is a predetermined parameter that we study in Figure 4.6c. Here again, the behavior of the maximum FCT does not follow a clear pattern with ϵ. The mean FCT however describes a similar pattern as other RSS schemes: as *espilon* increases, the rate at which flows are being rerouted also increases. This constitutes the reactive part of RSS, and therefore decreases the mean FCT. Past a threshold however, Random-Time RSSreaches a stable state, and higher rerouting probabilities do not have an effect anymore.

Sirona

To be sure to reroute the right subflow, Sirona needs at least one data point for the maximum RTT per subflow. In a fraction of the update interval of duration t, for a congestion window



(a) Symmetric

(b) Asymmetric

Figure 4.7: Comparison of the different RSS Flavors with Sirona.

w (in packets), and assuming no packet trims are received during t,

$$\sum_{c=w}^{w+\lfloor t/RTT \rfloor - 1} c$$

packets acknowledgments will be received by the sender. Thus, the condition can be written as

$$\sum_{c=w}^{w+\lfloor t/RTT \rfloor - 1} c > \sigma$$

We could set the value of t dynamically based on the current size of the congestion window, but default to a simpler, empirically motivated scheme where we pick t to be half of the update period, *i.e.* $t = 2RTT$. We define ϕ as the ratio of t to the update interval.

As shown in Figure 4.6d, mean, 99$^{\text{th}}$ percentile, and maximum FCT values all stabilize once about 35% of the update period is considered. At the highest value (1), all the update interval is taken into account to determine the worst-performing subflow.

4.4.5 RSS flavors

Using the lessons from 4.4.4, we evaluate the different versions of RSS against Sirona in Figure 4.7. As expected, QPRSS benefits from both the reactive effects of RSS and the added interventions of the congestion control algorithm on the subflow level. The number of subflows for QPRSS is the same as for other RSS schemes, giving it a significant advantage at the cost of significantly increased memory footprint (as described in 4.3.3). RTRSS suffers from too random rerouting decisions, rerouting subflows experiencing only transient delay, yielding up to 8% increase maximum FCT. Frozen RSS slightly suffers from the challenge of choosing the right threshold value, but stays within 5% of Sirona. Finally, Sirona and RSS display very similar behaviors, with less than 2% maximum FCT difference between them.

4.4.6 Metrics

The choice of the metric to determine the worst-performing subflow has little impact on the FCT. RSS is adaptable to various congestion signals: Vanilla RSS, Maximum RTT, mean RTT, EWMA, and Sirona are all within 3% of each other, both in terms of mean and max RTT, and both for symmetric and asymmetric setups. Therefore, looking at the worse RTT in Sirona allows us to save significant memory while having similar performances to RSS.

4.4.7 Flow size

RSS has two components: dividing flows into subflows, and reacting dynamically to congestion. When flows are short, RSS does not have time to react to congestion and only benefits from having more subflows. We plot in Figure 4.8 the maximum FCT for different flow sizes, in the symmetric setup. For the smallest flows (1 MTU), all the schemes are equivalent and therefore perform similarly. As the size of messages increases, Flowbender and ECMP suffer from a higher probability of flow collision. For small message sizes, USS equals RSS and Sirona's performance, showing that all the gains are indeed stemming from the subflow

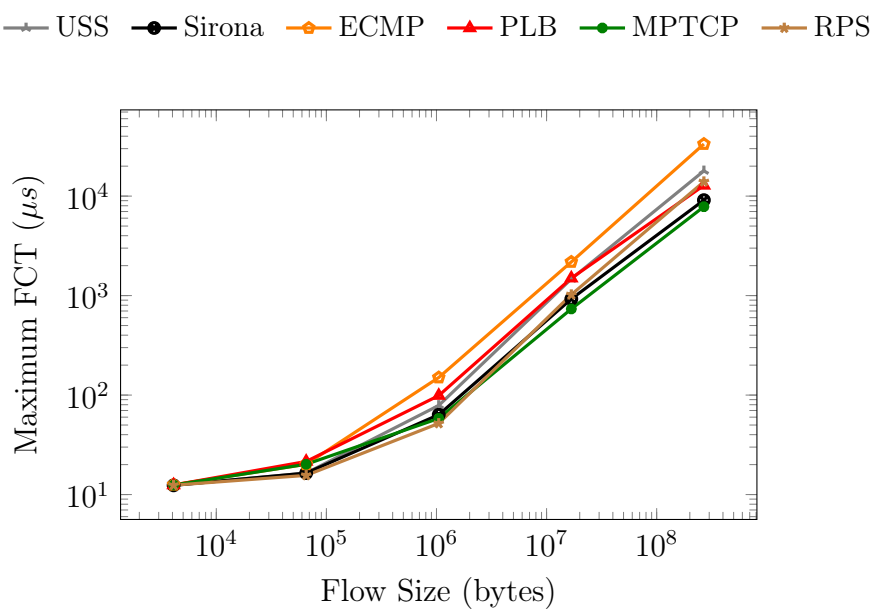
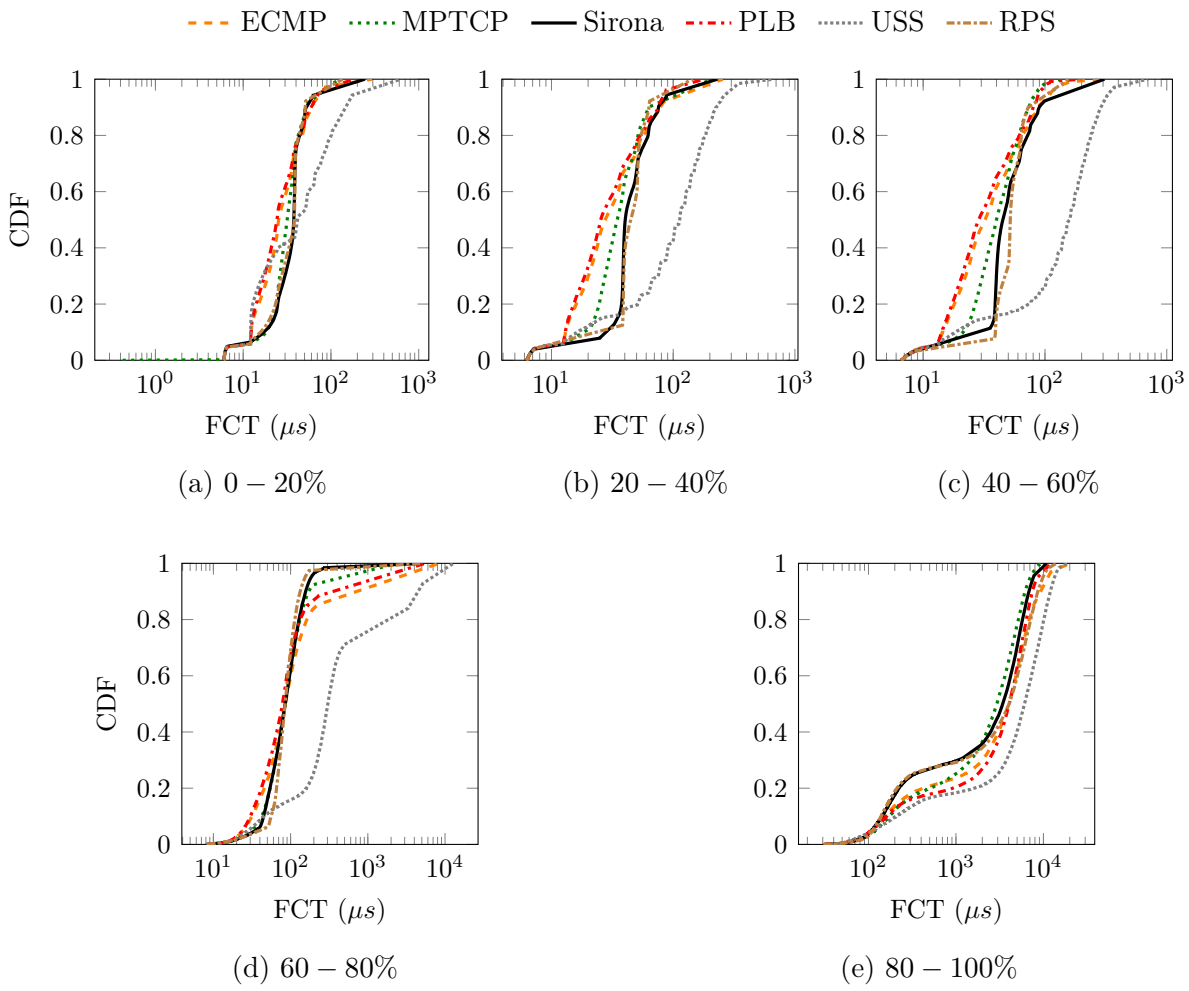


Figure 4.8: Impact of the flow size on FCT.

spraying part of RSS. While RPS initially benefits from finer granularity than RSS, as the message size increases and reactivity plays an increasing role in the final performance, RSS ends up outperforming RPS.

4.4.8 Web Search Traffic

Using the distribution in [143, Fig. 6.a.], we generate a web search traffic pattern by sampling from the same distribution and adjusting the interarrival between flows to obtain a load of 50% on the network. We then observe in Figure 4.9 the FCT, split by the flow size's quintile. For example, Figure 4.9a shows the distribution of the FCT for the 20% smallest flows. Overall, ECMP achieves 33% and 35% higher mean and max FCT than Sirona. Similarly, Flowbender, RPS, and USS respectively yield 20%, 26%, and 54% higher mean FCT, and 17%, 24%, and 49% higher max FCT. For smaller flows, the reactivity part of RSS or Sirona does not have time to get into action, and the only difference in schemes is the use of subflows. For the quintile with the highest flow sizes, RSS and Sirona can reroute flows away from congested paths, yielding extra benefits.



(a) $0 - 20\%$ (b) $20 - 40\%$ (c) $40 - 60\%$

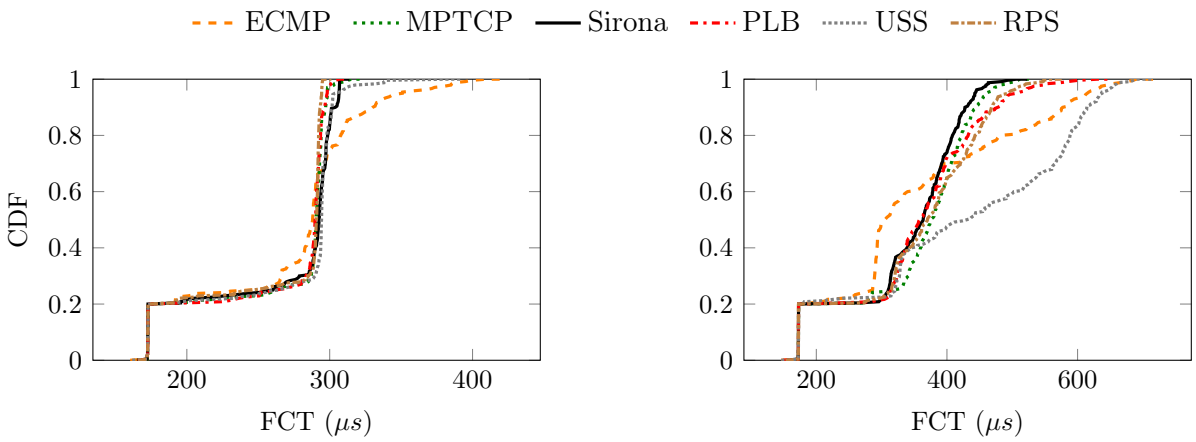
(d) $60 - 80\%$ (e) $80 - 100\%$

Figure 4.9: FCT distributions for a web search trace, split by quintile of flow size.

4.4.9 All to All traffic

We plot in Figure 4.10 the CDF of the FCT for the baseline schemes using an A2A traffic matrix for the symmetric (4.10a) and asymmetric (4.10b) topologies. For simulation time purposes, this experiment runs on a 16-node pod, with 4 6-port ToR switches and 2 4-port aggregate switches. The asymmetric topology has 1 aggregate switch with 50% capacity.

In the symmetric case, all schemes perform fairly similarly in mean FCT. Sirona also achieves similar performances in max FCT as Flowbender, RPS, and an MPTCP, and outperforms USS, and ECMP by 10%, 19%, and 25%. Compared to Figure 4.3a, MPTCP yields worse max FCT relative to Sirona, suggesting that the per-subflow congestion window



(a) Symmetric

(b) Asymmetric

Figure 4.10: FCT distribution for baseline schemes and Sirona for an A2A traffic pattern.

can suffer from outdated information.

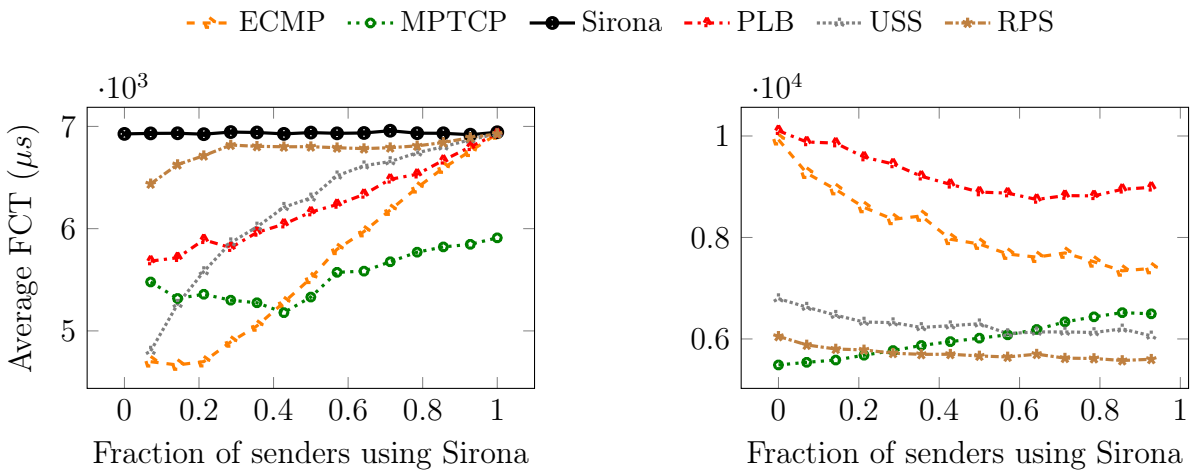
In the asymmetric case, for maximum FCT, Sirona fares similarly to MPTCP ($< 1\%$), and improves on RPS (9%), Flowbender (20%), USS (26%) and ECMP (27%).

4.4.10 Progressive Rollout

Datacenter clusters often require progressive rollout of updates both to minimize the impact of potential issues and to avoid complete unavailability of resources during the update.

To deploy a scheme like Sirona, a phase with senders using Sirona and senders using another load balancing scheme co-existing would be necessary. For every baseline scheme, we run an experiment with an increasingly large fraction of senders using Sirona, to simulate servers progressively adopting Sirona. We plot in Figure 4.11 the mean FCT averaged over senders using Sirona, and over senders using the other load balancing scheme.

Adding some Sirona servers in the cluster reduces the average FCT for all schemes but MPTCP: Sirona achieves better load balancing than most other schemes, and therefore removes some of the adversarial randomness negatively impacting performances. MPTCP only slightly suffers from sharing the network with Sirona traffic.



(a) Sirona Traffic

(b) Other Load-Balancing Scheme

Figure 4.11: Mixed Sirona/other scheme clusters

4.5 Related Work

We detail in this section related prior work in datacenter load balancing literature and point out significant differences with Sirona.

PLB [119] builds on top of Flowbender, which we compared RSS to. The differences between PLB and Flowbender lie in important implementation aspects (PLB uses the IPv6 Flow Label for entropy, and reroutes packets during idle periods instead of in the middle of flows to avoid issues with packet ordering). Our implementation of Flowbender does not particularly suffer from the issues motivating the upgrade from Flowbender to PLB (in particular, we assume support for out-of-order packets). Hermes [144] builds on top of Flowbender and extends the congestion signal to a mix of ECN and RTT. Therefore, most of the reasons for Sirona outperforming Flowbender hold when comparing Sirona to PLB or Hermes.

Similarly to Sirona, MPTCP [122] divides flows into subflows to take advantage of finer load balancing granularity. MPTCP subflows each have their own congestion window, similar to QPRSS. While QPRSS outperforms Sirona, the memory requirements and the implementation complexity of MPTCP [126] make it particularly challenging to implement

in hardware. On top of this, MPTCP remains less flexible than Sirona in terms of path selection: for example, MPTCP would be unable to use new paths appearing during the lifetime of a flow.

Presto [145] also relies on dividing flows into flowcells, or contiguous 64 kB flow segments. One advantage of this approach is to avoid re-ordering issues for mouse flows with flow sizes smaller than 64 kB. However, Presto deals with asymmetric topologies by taking load balancing decisions at an end-to-end level, significantly increasing the complexity of the system, and requiring end-point knowledge of the network topology. In [131], authors leverage flowlet switching (as introduced in [146] to divide TCP flows while avoiding packet reordering) to perform flow balancing. Practically, flows are divided into bursts of packets separated by an idle period, and new routing decisions are made during the idle periods. Flowlet switching benefits from lower granularity and elasticity in load balancing and minimizes packet reordering, but has been shown to react poorly to congestion [144]. On the other hand, Sirona is flexible to dynamic network topologies and requires minimal space or computation on end-hosts by leveraging the existing hashing capabilities of datacenter switches.

DRB [147] and RPS [123] both rely on per-packet load balancing granularity and therefore achieve good load balancing performances in symmetric scenarios. However, as shown in section 4.4, RPS adjusts its sending rate in asymmetric scenarios to the slowest link available. DRB circumvents this problem by giving all servers perfect knowledge of the network topology. DRILL [125] is also based on per-packet load balancing but focuses on switch-level congestion, and does not consider heterogeneous topologies. Instead, Sirona requires no knowledge of the topology, and can seamlessly adjust to failures or new paths.

CONGA [148] and Juniper VCF [149] both rely on specialized switch hardware to load balance the network. Hula [128] relaxes this requirement and builds on programmable hardware, but all three require modifications of the network switches, while Sirona only runs on end-hosts.

4.6 Summary

In this chapter, we introduce Sirona, a memory-efficient version of Reactive Subflow Spraying 4.2.2. Sirona is particularly suited for hardware deployments and is currently being implemented for potential deployment at a major datacenter company. We show through large-scale simulations that Sirona consistently achieves FCTs within a few percent of RSS, and generally exhibits similar behaviors in various scenarios. We also extend the analysis of RSS to study alternative schemes and provide insight to identify key performance parameters.

Chapter 5

Conclusion

This dissertation has presented three systems that address fundamental challenges in modern networks, from datacenter task scheduling to backbone network performance and load balancing efficiency. Through rigorous analysis and modeling, we have built principled approaches to networking systems, relying on combinatorial theory, queueing theory, and coding theory.

Nona introduces stochastic scheduling, to show the potential of randomized algorithms for datacenter task scheduling. Through a queueing theoretic model, we incorporate network queueing costs in the scheduling problem and show how the network awareness of Nona makes it outperform but multiple orders of magnitude traditional task scheduling solutions in constrained networking environments.

LINC points out the potential for backbone network operators of network coding solutions. By focusing on in-network interventions, we show that network coding approaches also successfully reduce retransmissions and effective network latency without needing end-host control. We build a rigorous mathematical model to understand the sources of LINC's gains and verify its assumptions in simulation.

Sirona extends previous work based on packet spraying and reactive load balancing schemes by reducing by 3× the memory footprint while keeping the load balancing capabilities of

sub-flow-level schemes and the adaptability of reactive schemes.

As network demands continue to grow and evolve, the need for innovative solutions across all layers of network infrastructure becomes increasingly critical. This dissertation demonstrates that significant improvements are possible through careful and explainable system design based on time-tested theoretical results.

References

[1] B. Pit-Claudel, D. Malak, A. Cohen, M. Medard, and M. Ghobadi. "Nona: A Stochastic Congestion-Aware Job Scheduler for Real-Time Inference Queries". In: *2024 IEEE 13th International Conference on Cloud Networking (CloudNet)*. 2024, pp. 1–10. DOI: 10.1109/CloudNet62863.2024.10815926.

[2] A. Ghalayini. "Using synchronized clocks to improve temporal fairness, congestion control, and load balancing in datacenter networks". PhD thesis. Stanford University, 2023.

[3] A. Gangidi et al. "RDMA over Ethernet for Distributed Training at Meta Scale". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 57–70. ISBN: 9798400706141. DOI: 10.1145/3651890.3672233. URL: https://doi.org/10.1145/3651890.3672233.

[4] A. Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 183–197. ISSN: 0146-4833. DOI: 10.1145/2829988.2787508. URL: https://doi.org/10.1145/2829988.2787508.

[5] Cisco. *Cisco global cloud index: Forecast and methodology, 2014–2019*. White Paper. 2015. URL: https://www.cisco.com/c/dam/m/en_us/service-provider/ciscoknowledgenetwork/files/547_11_10-15-DocumentsCisco_GCI_Deck_2014-2019_for_CKN__10NOV2015_.pdf.

[6] A. Andreyev. *Introducing data center fabric, the next-generation Facebook data center network*. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/. Nov. 2014.

[7] K. Qian et al. "Alibaba HPN: A Data Center Network for Large Language Model Training". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 691–706. ISBN: 9798400706141. DOI: 10.1145/3651890.3672265. URL: https://doi-org.libproxy.mit.edu/10.1145/3651890.3672265.

[8] Cisco. *Cisco Annual Internet Report (2018–2023)*. White Paper. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[9] A. Mauldin. *Used International Bandwidth Reaches New Heights*. 2024. URL: https://blog.telegeography.com/used-international-bandwidth-reaches-new-heights.

[10] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[11] L. Weng and G. Brockman. *Techniques for training large neural networks*. 2022.

[12] S. Gheissi, S. Mahmood, and S. Ghorbani. "Burstiness in Data Center Topologies". In: *Proc. of the 3rd International CoNEXT Student Workshop*. CoNEXT-SW '22. Rome, Italy: ACM, 2022, pp. 29–31. ISBN: 9781450399371. DOI: 10.1145/3565477.3569160.

[13] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. "Micro-Burst in Data Centers: Observations, Analysis, and Mitigations". In: *2018 IEEE 26th International Conference on Network Protocols*. 2018, pp. 88–98. DOI: 10.1109/ICNP.2018.00019.

[14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-resource packing for cluster schedulers". In: *Proc., ACM SIGCOMM Computer Communication Review*. Vol. 44. 2014, pp. 455–466.

[15] Z. Jia, M. Zaharia, and A. Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks". In: *Proc. of Machine Learning and Systems* 1 (2019), pp. 1–13.

[16] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. "Learning scheduling algorithms for data processing clusters". In: *Proc. of the ACM Special Interest Group on Data Communication*. Beijing, China, 2019, pp. 270–288. ISBN: 9781450359566.

[17] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proc. of the Thirteenth EuroSys Conference*. Porto, Portugal: ACM, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190517.

[18] W. Xiao et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3.

[19] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *Proc. of the 16th USENIX Conference on Networked Systems Design and Implementation*. Boston, MA, USA, 2019, pp. 485–500. ISBN: 9781931971492.

[20] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning". In: *15th USENIX Symp. on Operating Systems Design and Implementation*. July 2021, pp. 1–18. ISBN: 978-1-939133-22-9.

[21] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. "Themis: Fair and Efficient GPU Cluster Scheduling". In: *17th USENIX Symp. on Networked Systems Design and Implementation*. Santa Clara, CA, Feb. 2020, pp. 289–304. ISBN: 978-1-939133-13-7.

[22] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino. "Efficient Algorithms for Device Placement of DNN Graph Operators". In: *Advances*

in Neural Information Processing Systems. Vol. 33. Curran Associates, Inc., 2020, pp. 15451–15463.

[23] I. Grosof, Z. Scully, M. Harchol-Balter, and A. Scheller-Wolf. "Optimal Scheduling in the Multiserver-Job Model under Heavy Traffic". In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.3 (Dec. 2022). DOI: 10.1145/3570612.

[24] R. Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. Springer Science & Business Media, 2013.

[25] S. Feizi, M. Médard, and M. Effros. "Compressive sensing over networks". In: *Proc., Annual Allerton Conference on Communication, Control, and Computing*. Sept. 2010, pp. 1129–1136.

[26] Z. Li et al. "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving". In: *17th USENIX Symp. on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 663–679. ISBN: 978-1-939133-34-2.

[27] D. Bertsekas and R. Gallager. *Data networks (2nd ed.)* USA: Prentice-Hall, Inc., 1992. ISBN: 0132009161.

[28] A. Y. Khintchine. "Mathematical theory of a stationary queue". In: *Matematicheskii Sbornik* 39.4 (1932), pp. 73–84.

[29] F. Pollaczek. "Über eine Aufgabe der Wahrscheinlichkeitstheorie. I". In: *Math Z* 32 (1930), pp. 64–100. DOI: https://doi.org/10.1007/BF01194620.

[30] J. R. Jackson. "Jobshop-like Queueing Systems". In: *Management Science* 10.1 (1963), pp. 131–142. ISSN: 00251909, 15265501.

[31] M. Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

[32] X. Yao and A. Klimovic. *DeltaZip: Multi-Tenant Language Model Serving via Delta Compression*. 2023. arXiv: 2312.05215 [cs.DC].

[33] Y. Kochura, Y. Gordienko, V. Taran, N. Gordienko, A. Rokovyi, O. Alienin, and S. Stirenko. "Batch Size Influence on Performance of Graphic and Tensor Processing Units During Training and Inference Phases". In: *Advances in Computer Science for Engineering and Education II*. Ed. by Z. Hu, S. Petoukhov, I. Dychka, and M. He. Cham: Springer International Publishing, 2020, pp. 658–668. ISBN: 978-3-030-16621-2.

[34] A. Murray, D. Kim, and J. Combs. "The promise of a decentralized internet: What is Web3 and how can firms prepare?" In: *Business Horizons* 66.2 (2023), pp. 191–202. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2022.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0007681322000714.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. *Language Models are Unsupervised Multitask Learners*. 2019.

[36] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[37] A. Z. Karen Simonyan. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *Proc. of the 3rd International Conference on Learning Representations (ICLR)*. May 2015.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proc. of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada, 2012, pp. 1097–1105.

[39] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[40] M. Naumov et al. *Deep Learning Recommendation Model for Personalization and Recommendation Systems*. 2019. arXiv: 1906.00091 [cs.IR].

[41] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch. "TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs". In: *20th USENIX Symp. on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA, Apr. 2023, pp. 739–767. ISBN: 978-1-939133-33-5.

[42] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. July 2021, pp. 397–411. ISBN: 978-1-939133-23-6.

[43] H. Geng, D. Zeng, and Y. Li. "Performance Efficient Layer-aware DNN Inference Task Scheduling in GPU Cluster". In: *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 2022, pp. 2242–2247. DOI: 10.1109/GLOBECOM48099.2022.10001581.

[44] L. Fratta, M. Gerla, and L. Kleinrock. "The flow deviation method: An approach to store-and-forward communication network design". In: *Networks* 3.2 (1973), pp. 97–133. DOI: https://doi.org/10.1002/net.3230030202. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230030202.

[45] D. J. Daley and D. Vere-Jones. *An introduction to the theory of point processes: volume II: general theory and structure*. Springer, 2008.

[46] P. Sinha, A. Guliani, R. Jain, B. Tran, M. D. Sinclair, and S. Venkataraman. "Not All GPUs Are Created Equal: Characterizing Variability in Large-Scale, Accelerator-Rich Systems". In: *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022.

[47] L. Kleinrock. "Message delay in communication nets with storage". PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 1963.

[48] E. V. Morozov, I. V. Peshkova, and A. S. Rumyantsev. "Bounds and Maxima for the Workload in a Multiclass Orbit Queue". In: *Mathematics* 11.3 (Jan. 2023), p. 564.

[49] J. R. Jackson. "Networks of Waiting Lines". In: *Operations Research* 5.4 (1957), pp. 518–521. ISSN: 0030364X, 15265463. (Visited on 07/19/2024).

[50] C. E. Leiserson. "Fat-trees: Universal networks for hardware-efficient supercomputing". In: *IEEE Transactions on Computers* C-34.10 (1985), pp. 892–901. DOI: 10.1109/TC.1985.6312192.

[51] D. Berend and T. Tassa. "Efficient Bounds on Bell Numbers and on Moments of Sums of Random Variables". In: *Probability and Mathematical Statistics* 30 (Jan. 2010).

[52] F. Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[53] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[54] M. D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG].

[55] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[56] S. J. Reddi, S. Kale, and S. Kumar. *On the Convergence of Adam and Beyond*. 2019. arXiv: 1904.09237 [cs.LG].

[57] H. B. McMahan et al. "Ad Click Prediction: A View from the Trenches". In: *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, 2013, pp. 1222–1230. ISBN: 9781450321747. DOI: 10.1145/2487575.2488200.

[58] J. Nocedal and S. J. Wright. *Numerical Optimization*. New York, NY, USA: Springer, 1999.

[59] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proc. of the 27th ACM Symp. on Operating Systems Principles*. Huntsville, Ontario, Canada, 2019, pp. 1–15. ISBN: 9781450368735. DOI: 10.1145/3341301.3359646.

[60] W. Chun-Feng, L. San-Yang, and S. Pei-Ping. "Global optimization for sum of geometric fractional functions". In: *Applied Mathematics and Computation* 216.8 (2010), pp. 2263–2270. ISSN: 0096-3003.

[61] F. Bazikar and M. Saraj. "Solving fractional geometric programming problems via relaxation approach". In: *MatLAB J* 1.3 (2018), pp. 1–14.

[62] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen. "Deep Learning Workload Scheduling in GPU Datacenters: A Survey". In: *ACM Comput. Surv.* 56.6 (Jan. 2024). ISSN: 0360-0300. DOI: 10.1145/3638757.

[63] G. R. Gilman, S. S. Ogden, R. J. Walls, and T. Guo. "Challenges and Opportunities of DNN Model Execution Caching". In: *Proc. of the Workshop on Distributed Infrastructures for Deep Learning*. DIDL '19. Davis, CA, USA: ACM, 2019, pp. 7–12. ISBN: 9781450370370. DOI: 10.1145/3366622.3368147.

[64] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 9781450323888. DOI: 10.1145/2517349.2522716.

[65] Y. Huang et al. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism". In: *Proc. of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA, 2019.

[66] R. Viswanathan, G. Ananthanarayanan, and A. Akella. "CLARINET: WAN-Aware Optimization for Analytics Queries". In: *Proc., USENIX Symp. on Operating Systems Design and Implementation*. 2016, pp. 435–450.

[67] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation*. Broomfield, CO, 2014, pp. 285–300. ISBN: 9781931971164.

[68] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364.

[69] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters". In: *Proc., USENIX Symp. on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 81–97.

[70] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr. "A Fundamental Tradeoff Between Computation and Communication in Distributed Computing". In: *IEEE Trans. Inf. Theory* 64 (Jan. 2018), pp. 109–128.

[71] Y. H. Ezzeldin, M. Karmoose, and C. Fragouli. "Communication vs distributed computation: an alternative trade-off curve". In: *Proc., IEEE Inf. Theory Wkshp*. Kaohsiung, Taiwan: IEEE, Nov. 2017, pp. 279–283.

[72] M. Kiamari, C. Wang, and A. S. Avestimehr. "On heterogeneous coded distributed computing". In: *Proc., IEEE Globecom*. Marina Bay, Singapore, Dec. 2017, pp. 1–7.

[73] T. Ho, M. Medard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. "A Random Linear Network Coding Approach to Multicast". In: *IEEE Transactions on Information Theory* 52.10 (Oct. 2006), pp. 4413–4430. DOI: 10.1109/TIT.2006.881746.

[74] Q. Yu, M. Maddah-Ali, and S. Avestimehr. "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication". In: *Advances in Neural Information Processing Systems* 30 (2017).

[75] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr. "A fundamental tradeoff between computation and communication in distributed computing". In: *IEEE Trans. Inf. Theory* 64.1 (Sept. 2017), pp. 109–128.

[76] S. Li and S. Avestimehr. *Coded Computing: Mitigating Fundamental Bottlenecks in Large-Scale Distributed Computing and Machine Learning*. Vol. 17. 1. Foundation, Trends in Communications, and Information Theory, 2020, pp. 1–148.

[77] A. Khalesi and P. Elia. "Multi-user linearly-separable distributed computing". In: *IEEE Trans. Inf. Theory* (2023).

[78] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover. "On the optimal recovery threshold of coded matrix multiplication". In: *IEEE Transactions on Information Theory* 66.1 (2019), pp. 278–301.

[79] S. Dutta, H. Jeong, Y. Yang, V. Cadambe, T. M. Low, and P. Grover. "Addressing Unreliability in Emerging Devices and Non-von Neumann Architectures Using Coded Computing". In: *Proc. of the IEEE* 108.8 (May 2020), pp. 1219–34.

[80] D. Data, L. Song, and S. Diggavi. "Data encoding for Byzantine-resilient distributed optimization". In: *arXiv preprint arXiv:1907.02664* (2019).

[81] J. Brutlag. *Speed Matters for Google Web Search*. http://goo.gl/vJq1lx. Accessed: 2024-06-10. July 2015.

[82] S. Wang et al. "Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 113–129. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/wang-shibo.

[83] Akamai. *10for10*. https://web.archive.org/web/20180510215425/https://www.akamai.com/us/en/multimedia/documents/brochure/akamai-10for10-brochure.pdf. Accessed: 2024-06-10. July 2015.

[84] D. Bhattacherjee et al. "cISP: A Speed-of-Light Internet Service Provider". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1115–1133. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/bhattacherjee.

[85] Starlink. URL: https://www.starlink.com/us.

[86] Taara. *Expanding global access to fast, affordable internet with beams of light*. 2018.

[87] Amazon. *Project Kuiper*. URL: https://www.aboutamazon.com/what-we-do/devices-services/project-kuiper.

[88] Telesat. URL: https://www.telesat.com/.

[89] A. Myers, B. Nigito, and N. Foster. "Network Design Considerations for Trading Systems". In: *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. HotNets '24. Irvine, CA, USA: Association for Computing Machinery, 2024, pp. 282–289. ISBN: 9798400712722. DOI: 10.1145/3696348.3696890. URL: https://doi.org/10.1145/3696348.3696890.

[90] T. Bai, A. Alkhateeb, and R. W. Heath. "Coverage and capacity of millimeter-wave cellular networks". In: *IEEE Communications Magazine* 52.9 (2014), pp. 70–77. DOI: 10.1109/MCOM.2014.6894455.

[91] K. Jain, J. Padhye, V. N. Padmanabhan, and L. Qiu. "Impact of interference on multi-hop wireless network performance". In: *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*. MobiCom '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 66–80. ISBN: 1581137532. DOI: 10.1145/938985.938993. URL: https://doi.org/10.1145/938985.938993.

[92] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill. "Run, Walk, Crawl: Towards Dynamic Link Capacities". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets '17. Palo Alto, CA, USA: Association for Computing Machinery, 2017, pp. 143–149. ISBN: 9781450355698. DOI: 10.1145/3152434.3152451. URL: https://doi.org/10.1145/3152434.3152451.

[93] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill. "RADWAN: rate adaptive wide area network". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 547–560. ISBN: 9781450355674. DOI: 10.1145/3230543.3230570. URL: https://doi.org/10.1145/3230543.3230570.

[94] M. Ghobadi and R. Mahajan. "Optical Layer Failures in a Large Backbone". In: *Proceedings of the 2016 Internet Measurement Conference*. IMC '16. Santa Monica, California, USA: Association for Computing Machinery, 2016, pp. 461–467. ISBN: 9781450345262. DOI: 10.1145/2987443.2987483. URL: https://doi.org/10.1145/2987443.2987483.

[95] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. "The Internet Topology Zoo". In: *IEEE Journal on Selected Areas in Communications* 29.9 (2011), pp. 1765–1775. DOI: 10.1109/JSAC.2011.111002.

[96] L. Schumann, T. V. Doan, T. Shreedhar, R. Mok, and V. Bajpai. *Impact of Evolving Protocols and COVID-19 on Internet Traffic Shares*. 2022. arXiv: 2201.00142 [cs.NI]. URL: https://arxiv.org/abs/2201.00142.

[97] J. Krigslund, J. Hansen, D. E. Lucani, F. H. P. Fitzek, and M. Medard. "Network Coded Software Defined Networking: Design and Implementation". In: *Proceedings of European Wireless 2015; 21th European Wireless Conference*. 2015, pp. 1–6.

[98] T. Ho, M. Medard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong. "A Random Linear Network Coding Approach to Multicast". In: *IEEE Transactions on Information Theory* 52.10 (2006), pp. 4413–4430. DOI: 10.1109/TIT.2006.881746.

[99] V. Adat Vasudevan, T. Soni, and M. Médard. "Practical Sliding Window Recoder: Design, Analysis, and Usecases". In: July 2023, pp. 1–6. DOI: 10.1109/LANMAN58293.2023.10189420.

[100] P. Enenche, D. H. Kim, and D. You. "Network Coding as Enabler for Achieving URLLC Under TCP and UDP Environments: A Survey". In: *IEEE Access* 11 (2023), pp. 76647–76674. DOI: 10.1109/ACCESS.2023.3297137.

[101] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros. "Network Coding Meets TCP: Theory and Implementation". In: *Proceedings of the IEEE* 99.3 (2011), pp. 490–512. DOI: 10.1109/JPROC.2010.2093850.

[102] M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, D. Leith, and M. Medard. *Network Coded TCP (CTCP)*. 2013. arXiv: 1212.2291 [cs.NI]. URL: https://arxiv.org/abs/1212.2291.

[103] A. Cohen, D. Malak, V. B. Bracha, and M. Médard. "Adaptive Causal Network Coding With Feedback". In: *IEEE Transactions on Communications* 68.7 (2020), pp. 4325–4341. DOI: 10.1109/TCOMM.2020.2989827.

[104] A. Cohen, G. Thiran, V. Bracha, and M. Médard. "Adaptive Causal Network Coding with Feedback for Multipath Multi-hop Communications". In: June 2020, pp. 1–7. DOI: 10.1109/ICC40277.2020.9149421.

[105] Intel. *Tofino Series*. URL: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[106] S. Hasan et al. "Building Flexible, Low-Cost Wireless Access Networks With Magma". In: *20th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 23). Boston, MA: USENIX Association, Apr. 2023, pp. 1667–1681. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/hasan.

[107] Facebook. *Introducing Facebook's new terrestrial connectivity systems — Terragraph and Project ARIES*. URL: https://engineering.fb.com/2016/04/13/connectivity/introducing-facebook-s-new-terrestrial-connectivity-systems-terragraph-and-project-aries/.

[108] HughesNet. *Satellite Internet to Connect, Stream and Play*. URL: https://www.hughesnet.com/.

[109] viaSat. *Global Communications*.

[110] S. Paul, E. Ayanoglu, T. La Porta, K.-W. Chen, K. Sabnani, and R. Gitlin. "An asymmetric protocol for digital cellular communications". In: *Proceedings of INFOCOM'95*. Vol. 3. 1995, 1053–1062 vol.3. DOI: 10.1109/INFCOM.1995.515982.

[111] F. Lefevre and G. Vivier. "Understanding TCP's behavior over wireless links". In: *IEEE Benelux Chapter on Vehicular Technology and Communications. Symposium on Communications and Vehicular Technology. SCVT-2000. Proceedings (Cat. No.00EX465)*. 2000, pp. 123–130. DOI: 10.1109/SCVT.2000.923350.

[112] T. Lakshman and U. Madhow. "The performance of TCP/IP for networks with high bandwidth-delay products and random loss". In: *IEEE/ACM Transactions on Networking* 5.3 (1997), pp. 336–350. DOI: 10.1109/90.611099.

[113] S. Fahmy, V. Prabhakar, S. R. Avasarafa, and O. M. Younis. *TCP over Wireless Links: Mechanisms and Implications*. Tech. rep. Purdue University, Department of Computer Science, 2003.

[114] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. K. Ramakrishnan. "LT-TCP: End-to-End Framework to Improve TCP Performance over Networks with Lossy Channels". In: *Quality of Service – IWQoS 2005*. Ed. by H. de Meer and N. Bhatti.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 81–93. ISBN: 978-3-540-31659-6.

[115] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung. "Network information flow". In: *IEEE Transactions on Information Theory* 46.4 (2000), pp. 1204–1216. DOI: 10.1109/18.850663.

[116] M. v. d. Schaar and P. A. Schaar Mihaela van derChou. *Multimedia over IP and wireless networks*. Includes bibliographical references and index. . Burlington: Elsevier, 2007.

[117] S. Lin and D. J. Costello. *Error Control Coding*. Prentice Hall, 1983.

[118] C. Hopps. *RFC2992: Analysis of an Equal-Cost Multi-Path Algorithm*. USA, 2000.

[119] M. A. Qureshi, Y. Cheng, Q. Yin, Q. Fu, G. Kumar, M. Moshref, J. Yan, V. Jacobson, D. Wetherall, and A. Kabbani. "PLB: congestion signals are simple and effective for network load balancing". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 207–218. ISBN: 9781450394208. DOI: 10.1145/3544216.3544226. URL: https://doi.org/10.1145/3544216.3544226.

[120] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*. San Jose, CA: USENIX Association, Apr. 2010. URL: https://www.usenix.org/conference/nsdi10-0/hedera-dynamic-flow-scheduling-data-center-networks.

[121] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: Association for Computing Machinery, 2010, pp. 63–74. ISBN: 9781450302012. DOI: 10.1145/1851182.1851192. URL: https://doi.org/10.1145/1851182.1851192.

[122] O. Bonaventure, C. Paasch, and G. Detal. *Use Cases and Operational Experience with Multipath TCP*. RFC 8041. Jan. 2017. DOI: 10.17487/RFC8041. URL: https://www.rfc-editor.org/info/rfc8041.

[123] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. "On the impact of packet spraying in data center networks". In: *2013 Proceedings IEEE INFOCOM*. 2013, pp. 2130–2138. DOI: 10.1109/INFCOM.2013.6567015.

[124] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. "Juggler: a practical reordering resilient network stack for datacenters". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342407. DOI: 10.1145/2901318.2901334. URL: https://doi.org/10.1145/2901318.2901334.

[125] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. "DRILL: Micro Load Balancing for Low-latency Data Center Networks". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 225–238. ISBN: 9781450346535. DOI: 10.1145/3098822.3098839. URL: https://doi.org/10.1145/3098822.3098839.

[126] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. "FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 149–160. ISBN: 9781450332798. DOI: 10.1145/2674005.2674985. URL: https://doi.org/10.1145/2674005.2674985.

[127] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. "Clove: Congestion-Aware Load Balancing at the Virtual Edge". In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*.

CoNEXT '17. Incheon, Republic of Korea: Association for Computing Machinery, 2017, pp. 323–335. ISBN: 9781450354226. DOI: 10.1145/3143361.3143401. URL: https://doi.org/10.1145/3143361.3143401.

[128] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. "HULA: Scalable Load Balancing Using Programmable Data Planes". In: *Proceedings of the Symposium on SDN Research*. SOSR '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016. ISBN: 9781450342117. DOI: 10.1145/2890955.2890968. URL: https://doi.org/10.1145/2890955.2890968.

[129] M. Li, D. Ganesan, and P. Shenoy. "PRESTO: Feedback-driven Data Management in Sensor Networks". In: *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*. San Jose, CA: USENIX Association, May 2006. URL: https://www.usenix.org/conference/nsdi-06/presto-feedback-driven-data-management-sensor-networks.

[130] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. "Scalable, optimal flow routing in datacenters via local link balancing". In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: Association for Computing Machinery, 2013, pp. 151–162. ISBN: 9781450321013. DOI: 10.1145/2535372.2535397. URL: https://doi.org/10.1145/2535372.2535397.

[131] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. "Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 407–420. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini.

[132] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. "DeTail: reducing the flow completion time tail in datacenter networks". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Com-*

puter Communication. SIGCOMM '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 139–150. ISBN: 9781450314190. DOI: 10.1145/2342356.2342390. URL: https://doi.org/10.1145/2342356.2342390.

[133] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury. "Resilient Datacenter Load Balancing in the Wild". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 253–266. ISBN: 9781450346535. DOI: 10.1145/3098822.3098841. URL: https://doi.org/10.1145/3098822.3098841.

[134] D. Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[135] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[136] M. Raab and A. Steger. ""Balls into Bins" — A Simple and Tight Analysis". In: *Randomization and Approximation Techniques in Computer Science*. Ed. by M. Luby, J. D. P. Rolim, and M. Serna. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 159–170. ISBN: 978-3-540-49543-7.

[137] N. L. Johnson and S. Kotz. "Urn Models and Their Application: An Approach to Modern Discrete Probability Theory." In: *Biometrics* 34 (1978), p. 528. URL: https://api.semanticscholar.org/CorpusID:125786093.

[138] V. F. Kolchin, B. A. Sevast'yanov, and V. P. Chistyakov. "Random Allocations". In: (1978).

[139] K. Ramakrishnan, S. Floyd, and D. Black. *RFC3168: The Addition of Explicit Congestion Notification (ECN) to IP*. USA, 2001.

[140] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag. "A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC". In: *IEEE Micro* 40.6 (2020), pp. 67–73. DOI: 10.1109/MM.2020.3016891.

[141] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. "Re-architecting datacenter networks and stacks for low latency and high performance". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 29–42. ISBN: 9781450346535. DOI: 10.1145/3098822.3098825. URL: https://doi.org/10.1145/3098822.3098825.

[142] U. E. Consortium. *Ultra Ethernet Specification Update*. URL: https://ultraethernet.org/ultra-ethernet-specification-update/.

[143] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. "Inside the Social Network's (Datacenter) Network". In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 123–137. ISSN: 0146-4833. DOI: 10.1145/2829988.2787472. URL: https://doi.org/10.1145/2829988.2787472.

[144] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury. "Resilient Datacenter Load Balancing in the Wild". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 253–266. ISBN: 9781450346535. DOI: 10.1145/3098822.3098841. URL: https://doi.org/10.1145/3098822.3098841.

[145] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. "Presto: Edge-based Load Balancing for Fast Datacenter Networks". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 465–478. ISBN: 9781450335423. DOI: 10.1145/2785956.2787507. URL: https://doi.org/10.1145/2785956.2787507.

[146] S. Kandula, D. Katabi, S. Sinha, and A. Berger. "Dynamic load balancing without packet reordering". In: *SIGCOMM Comput. Commun. Rev.* 37.2 (Mar. 2007), pp. 51–62. ISSN: 0146-4833. DOI: 10.1145/1232919.1232925. URL: https://doi.org/10.1145/1232919.1232925.

[147] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. "Per-packet load-balanced, low-latency routing for clos-based data center networks". In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: Association for Computing Machinery, 2013, pp. 49–60. ISBN: 9781450321013. DOI: 10.1145/2535372.2535375. URL: https://doi.org/10.1145/2535372.2535375.

[148] M. Alizadeh et al. "CONGA: distributed congestion-aware load balancing for datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 503–514. ISBN: 9781450328364. DOI: 10.1145/2619239.2626316. URL: https://doi.org/10.1145/2619239.2626316.

[149] D. R. Hanks. *Juniper QFX5100 Series: A Comprehensive Guide to Building Next-Generation Networks*. O'Reilly Media, Inc, 2014.