Delivering Predictable Tail Latency in Data Center Networks

Kevin Zhao

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Thomas E. Anderson, Chair
Ratul Mahajan
Simon Peter

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Delivering Predictable Tail Latency in Data Center Networks

Kevin Zhao

Chair of the Supervisory Committee:
Thomas E. Anderson
Paul G. Allen School of Computer Science & Engineering

Modern web services decompose a user request into thousands of RPCs whose slowest 1% dominate end-to-end latency, costing revenue and straining user patience. Operators codify expectations as tail latency SLOs, but meeting them is difficult even in well-run data center networks. Although such networks expose configuration parameters that have a large impact on tail latency, like switch weights, congestion windows, and switch marking thresholds, operators typically set these parameters once and rarely revisit them. When workload characteristics shift, for example in burstiness, traffic mix, or demand patterns, the resulting mismatch between the workload and the network can degrade user-observed performance and cause SLO violations, even in networks that deploy congestion control, traffic engineering, and class-based scheduling. A natural response is to adapt network parameters when workloads change, but existing methods adjust parameters by trial and error, risking intermediate violations and slow convergence in high-dimensional, noisy settings.

This dissertation argues that *prediction-guided control* is an effective technique for delivering predictable tail latency in data center networks. It makes two contributions. First, `Parsimon` is a scalable tail-latency estimator. Through a series of approximations, `Parsimon` decouples links and simulates them in parallel, allowing it to run orders of magnitude faster than full-fidelity simulators while retaining distribution-level accuracy. Second, `Polyphony` embeds such estimators in a closed loop control system to improve network performance. It treats predictions as priors, fuses them with live measurements, and searches safely inside a trust region that resets as conditions drift. In a small testbed on real machines, `Polyphony` meets tail latency SLOs within minutes, whereas a state-of-the-art model-free tuner fails to converge after an hour.

Together, fast prediction and prediction-guided control form a promising toolkit for steering large networks toward better performance for latency-sensitive applications, reducing the cost of provisioning and the risk of unsafe exploration.

To my family

ACKNOWLEDGMENTS

It has been a privilege to be advised by Tom Anderson. When I came to graduate school not knowing what I wanted to study, Tom provided the guidance and encouragement I needed to find my way. Through the years, he has been a constant source of wisdom and inspiration. His remarkable breadth and depth of knowledge as an advisor are matched only by his fundamental warmth and kindness as a mentor. When I would go to him for advice, he would always listen carefully and try very hard to figure out what I wanted to do. Then, he would tell me to do that thing.

As if I didn't have enough to be grateful to Tom for, about a year into the program, he introduced me to Mohammad Alizadeh and Prateesh Goyal, and so began the collaboration that would shape my graduate school career. For all his gentle demeanor, Mohammad is an intellectual giant, bringing to every technical discussion sharp insights and rare clarity. He is, to me, an intellectual role model.

Prateesh was a friend and mentor. He combined a fierce intellect with a disarming friendliness, and he was quick to dispense advice that I usually took. Sometime later, Chenning Li joined Mohammad's group, and we had a fruitful collaboration that lasted the rest of my time in graduate school. I am happy to call him my friend, and thankful that he pushed me to hit my deadlines.

Thank you to the other members of my committee: Ratul Mahajan and Simon Peter, whose questions and comments put me on the spot but always made the work better. I am also grateful to Brent Stephens for hosting me as an intern and student researcher at Google—an experience that taught me a great deal.

I have had the pleasure of working with excellent undergraduate and master's students. To Anna Goncharenko, David Dai, Sid Lakshmanan, and Claire Li: thank you for making research fun. Many of my fondest memories of the last couple years are of you guys, and I doubt I would have made it to the end in one piece without you. I can't wait to see what you go on to do.

I am grateful to my friends in the systems lab, especially Niel Lebeck, Katie Lim, and Lequn Chen. Thank you for putting up with my silliness and my antics for all these years.

The staff at the Allen School have been indispensable to my actually graduating. Thank you in particular to Elise Dorough and Joe Eckert, for helping me navigate the intricacies of graduation requirements, patiently weathering the onslaught of questions, and patching things up when I inevitably bungle them.

Thank you to Henry Hoffmann for taking me on while I was an undergraduate who didn't know anything. You'll find your influence in these pages too.

And lastly, I am thankful for the love and support of my family: my parents, Juan Fang and Liyan Zhao; my big brother, Derek; my partner, Lola Pickle Wallace; and our cats, Yuki and Bao Bao Zhao.

CONTENTS

GLOSSARY xiii

1 INTRODUCTION 1
- 1.1 Constructing a fast tail latency estimator 4
- 1.2 Building a prediction-guided network controller 5
- 1.3 Published work 6
- 1.4 Synopsis and outline 6

2 BACKGROUND 9
- 2.1 Classical quality of service 9
  - 2.1.1 Per-flow fairness 9
  - 2.1.2 Deterministic bounds, traffic shaping, and per-flow reservation 10
  - 2.1.3 Class-based QoS 11
  - 2.1.4 Legacy and limitations 12
- 2.2 Capacity planning and traffic engineering 13
- 2.3 Congestion control 14
- 2.4 In-network flow scheduling transports 16
- 2.5 Admission and overload control 18
- 2.6 Model-free autotuners 18
- 2.7 Emerging fast predictors 20

3 SCALABLE TAIL LATENCY ESTIMATION 23
- 3.1 Introduction 23
- 3.2 Parsimon overview 26
- 3.3 Key methods: decompose and aggregate 29
  - 3.3.1 Generating link-level workloads 29
  - 3.3.2 Generating link-level topologies 29
  - 3.3.3 Post-processing link-level results 32
  - 3.3.4 Aggregating link-level estimates 34
  - 3.3.5 Primary source of speedup 35
  - 3.3.6 Primary sources of error 35
- 3.4 Complementary methods 38
  - 3.4.1 Fast link-level simulation 38
  - 3.4.2 Clustering and pruning simulations 38
- 3.5 Evaluation 40
  - 3.5.1 General setup 41
  - 3.5.2 Analysis on a large-scale network 43
  - 3.5.3 Sensitivity analysis at small scale 44
  - 3.5.4 Analysis of one configuration 48
  - 3.5.5 Mixed workloads 50
  - 3.5.6 Link failures 50
  - 3.5.7 Studying error sources 52
- 3.6 Conclusion 56

4 PREDICTION-GUIDED CONTROL 59

4.1 Introduction 59
4.2 Background and motivation 60
4.2.1 Existing methods 62
4.3 Polyphony overview 64
4.4 Safe prediction-guided optimization 65
4.4.1 Definitions and problem formulation 66
4.4.2 Modeling residuals 67
4.4.3 Bayesian optimization over the surrogate 68
4.4.4 Safe exploration 68
4.5 Adaptation and data quality 70
4.5.1 Adapting to changing g 70
4.5.2 Processing data samples 70
4.6 Measuring workloads online 71
4.6.1 Overview 71
4.6.2 Per-host probing 72
4.6.3 Per-rack sampling 73
4.6.4 Expanding samples into workloads 73
4.7 Limitations 74
4.8 Implementation 75
4.9 Evaluation 76
4.9.1 Setup 77
4.9.2 Convergence study 78
4.9.3 Adaptation study 80
4.9.4 Ablation study 83
4.9.5 Scalability analysis in ns-3 84
4.10 Related work 86
4.10.1 Tuners and controllers 86
4.10.2 Simulation-based optimization and control 87
4.11 Conclusion 87
5 DISCUSSION 89
5.1 Summary of limitations 89
5.1.1 `Parsimon`'s limitations 89
5.1.2 `Polyphony`'s limitations 91
5.2 Future work 92
5.3 Broader implications 95
6 CONCLUSION 97

BIBLIOGRAPHY 99

LIST OF FIGURES

Figure 3.1 CDF of ns-3 versus `Parsimon` for flow completion time slowdown across multiple flow size ranges, zoomed into the tail 25
Figure 3.2 Overview of `Parsimon` 27
Figure 3.3 An illustration of `Parsimon`'s workflow 27
Figure 3.4 An illustration of how `Parsimon` generates link-level topologies 29
Figure 3.5 An illustration of how `Parsimon` aggregates link-level results into a path-level point estimate 34
Figure 3.6 Workloads from Roy *et al.*'s study of Meta's data center network 41
Figure 3.7 CDFs of FCT slowdown estimated by ns-3 and two `Parsimon` variants 43
Figure 3.8 CDFs of p99 error between `Parsimon` and ns-3 across all scenarios drawn from the sample space in Table 3.3 45
Figure 3.9 Distributions of p99 error between `Parsimon` and ns-3, faceted by different workload and topology parameters 46
Figure 3.10 CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the scenario whose error is at the 85^{th} percentile of the p99 error distribution 48
Figure 3.11 CDFs of FCT slowdown for ns-3 and `Parsimon`, bucketed by workload and flow size 51
Figure 3.12 Errors between ns-3 and `Parsimon` in estimated FCT slowdowns when there is a link failure 52
Figure 3.13 The parking lot topology used in §3.5.7 52
Figure 3.14 CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic, both with and without cross traffic 53
Figure 3.15 CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic with regular or identical cross traffic 54
Figure 3.16 CDFs of FCT slowdown for the same scenario as in Fig. 3.15b, but with bursty cross traffic 55
Figure 4.1 Model-free Bayesian optimization leads to poor behavior during convergence 61

Figure 4.2 Model error and its consequences for three classes with varying SLOs 62
Figure 4.3 `Polyphony`'s closed-loop architecture 64
Figure 4.4 An illustration of Bayesian optimization with a residual surrogate and a trust region 65
Figure 4.5 Proposed pipeline for workload collection 71
Figure 4.6 Per-class 99th percentile flow completion time slowdowns under high SLO constraints 78
Figure 4.7 Objective over time under high SLO constraints 79
Figure 4.8 Parameter trajectories for `pol/m3` under tight SLO constraints 79
Figure 4.9 Per-class 99th percentile flow completion time slowdowns under workload shifts 81
Figure 4.10 Objective over time under regime shifts 82
Figure 4.11 Parameter evolution under `pol/m3` 82
Figure 4.12 Per-class 99th percentile slowdowns and switch weights under ablation 83
Figure 4.13 Global objective over time under component ablations 84
Figure 4.14 CDF of convergence time across ns-3 scenarios 85
Figure 4.15 `pol/m3`'s behavior in ns-3 86

LIST OF TABLES

Table 3.1 The `Parsimon` variants under consideration 42
Table 3.2 Running times and speed-up of `Parsimon` variants 43
Table 3.3 The sample space for the sensitivity analysis in §3.5.3 45
Table 3.4 The five scenarios with the highest error values from the sensitivity analysis in §3.5.3 47
Table 3.5 Prediction error of `Parsimon`/ns-3 for estimated p99 FCT slowdown 49
Table 3.6 The three workloads mixed together in §3.5.5 50
Table 4.1 Controller variants used in the evaluation 76
Table 4.2 Convergence metrics across SLO tightness levels in CloudLab 80
Table 4.3 Traffic load profiles used in the adaptation study 81

GLOSSARY

BDP Bandwidth-Delay Product. The product of a link's capacity and its round-trip time, representing the amount of data in flight.

CDF Cumulative Distribution Function. A function that gives the probability that a random variable is less than or equal to a given value. In networking, CDFs are commonly used to visualize latency or flow completion time distributions.

CLOSED-LOOP A control strategy where the controller continuously monitors performance and adjusts configurations in response, forming a feedback loop between measurement and control.

CLOUDLAB A flexible, scientific infrastructure for cloud computing research. Provides bare-metal access to computing resources for experimental research.

DCTCP Data Center TCP. A congestion control protocol designed for data center networks that uses Explicit Congestion Notification (ECN) to respond to network congestion.

DSCP Differentiated Services Code Point. A field in IP packet headers used to classify and mark traffic for different quality of service treatments.

ECMP Equal-Cost Multi-Path routing. A routing strategy that distributes traffic across multiple paths with equal cost.

ECN Explicit Congestion Notification. A mechanism that allows routers to signal congestion to endpoints by marking packets instead of dropping them.

EI Expected Improvement. An acquisition function used in Bayesian optimization to balance exploration and exploitation.

FCT Flow Completion Time. The total time taken for a flow to complete from start to finish.

GP Gaussian Process. A non-parametric Bayesian approach to machine learning used for regression and modeling.

GPS Generalized Processor Sharing. An idealized scheduling discipline that allocates bandwidth to flows in infinitesimally small increments, providing perfect fairness and minimum rate guarantees.

NS-3 Network Simulator 3. A discrete-event, packet-level network simulator used for research and education.

OPEN-LOOP A control strategy where the controller selects a configuration based only on offline predictions or prior measurements, without real-time feedback.

P99 99th percentile. The value below which 99% of observations fall. A common tail latency metric indicating that only 1% of requests experience worse performance.

RPC Remote Procedure Call. A protocol that allows a program to execute a procedure on another computer.

RTT Round-Trip Time. The time it takes for a packet to travel from sender to receiver and back, including propagation delay, transmission delay, queueing delay, and processing delay.

SLI Service-Level Indicator. A metric used to track service performance.

SLO Service-Level Objective. A predicate over a metric that defines the target performance level for a service.

SLOWDOWN Flow Completion Time normalized to the time on an unloaded network.

TAIL LATENCY The latency experienced by the slowest requests in a distribution, typically measured at high percentiles such as the 95th, 99th, or 99.9th percentile. In distributed systems, tail latency often dominates end-to-end performance due to fan-out patterns.

1

INTRODUCTION

Modern data center networks operate at enormous scale. Google's fifth-generation Jupiter network delivers 13 petabits per second of bisection bandwidth inside a single warehouse-scale facility [83]. To serve billions of users, large-scale applications are decomposed into thousands of microservices, which in turn issue billions of internal remote procedure calls[1] per second atop the underlying network [39]. Meta and Amazon report similar volumes [75], with Amazon's IAM authentication system alone processing over half a billion requests per second [20]. These systems underpin the low latency, highly available applications that we depend on every day.

Unfortunately, large scale also amplifies performance variability. As Dean and Barroso point out, large systems parallelize work across many machines, which can reduce mean latency but can also worsen the tail of the end-to-end latency distribution (i. e., *tail latency*), because a request must wait for the slowest of its subcalls [27]. Even rare slowdowns can dominate performance when requests fan out to hundreds or thousands of servers. For instance, even if only one in ten thousand backend calls takes over a second, a request that performs two thousand backend calls, even in parallel, still has a one in five chance of taking longer than one second [27]. Far from being a benign inconvenience, this effect directly translates into lost revenue. Google's own experiments show that adding 400 ms of delay to search results reduces the number of searches each user performs by 0.74% [22]. Similarly, injecting a 500 ms delay into the Bing search engine lowered revenue per user by 1.2% [85]. Put simply, sub-second delays can cost service providers billions of dollars.

Tail latency dominates user experience.

To mitigate this risk, operators define *service-level objectives* (SLOs) that track metrics such as the 99th percentile latency. An SLO is a predicate over some metric (called the *service-level indicator*, or SLI) making explicit the performance a system is expected to maintain [65]. For example, Microsoft's globally distributed database guarantees that reads and writes are served within 10 ms at the 99th percentile [10]. Here, tail latency is the SLI, and the guarantee about tail latency being under some bound is the SLO.

Operators codify expectations as SLOs.

While SLOs define clear performance targets, achieving them at large scale presents multiple challenges. Data center applications are often very latency sensitive [12, 34, 61, 97], with inherent dynamic variability in traffic matrices [70], a predominance of short flows [41],

1 A remote procedure call (RPC) invokes a function on another machine over the network, abstracting the underlying communication between microservices.

and frequent incast and outcast traffic patterns [79]. Link oversubscription (to reduce costs) and inevitable hardware failures further complicate performance management. Together, these characteristics make it difficult to predict when and where performance bottlenecks will emerge, hindering efforts to maintain consistent SLO compliance. In response, operators often allocate servers or bandwidth for peak loads (i. e., overprovisioning), which ensures compliance but wastes resources. Many systems also rely on static configurations, but fixed, hand-tuned settings that were once suitable may lose relevance over time as traffic patterns shift, workloads evolve, or deployments differ. Sustaining SLOs efficiently in the face of such dynamism requires systems to respond swiftly and efficiently to changing conditions.

In this work, we consider how to achieve tail latency SLOs in data center networks in an efficient, effective, and adaptive manner—*efficient*, by reducing wasteful overprovisioning where possible, *effective*, by ensuring that SLOs are consistently met, and *adaptive*, by tracking SLOs when network conditions change.

In our setting, we suppose that operators divide traffic into a small number of classes, each governed by an SLO, and we define latency at the granularity of a network flow, which we identify by the standard TCP connection five-tuple (source IP, source port, destination IP, destination port, and protocol). This flow-level view matches how applications experience network delay, as messages can only be processed once all packets have been delivered. For practical deployability, we restrict ourselves to mechanisms available in commodity switching hardware.

Improving network quality of service has been a longstanding goal, with decades of research producing a rich design space of mechanisms, from per-flow reservations to class-based differentiation, fair queueing, traffic shaping, admission control, and more. Chapter 2 surveys this history in detail. In production data center networks today, operators primarily rely on three techniques, albeit with mixed results. First, aggressive congestion control algorithms are designed to react quickly (within a few round trips) and forcefully to observed congestion along a path in an attempt to keep queues small [6, 7, 54, 58]. Second, on longer timescales, traffic engineering (such as weighted ECMP and Optical Circuit Switching) attempts to balance traffic volumes over available paths despite non-uniform topologies and inherent temporal and spatial variability in traffic demand [70]. Third, as a final backstop, mission-critical or latency-sensitive traffic is assigned to separate traffic classes to isolate it from other competing traffic, typically using scheduling weights rather than priorities to avoid starvation for low-priority traffic. Even with these mechanisms in place, well-managed data center networks still experience significant tail slowdowns for RDMA and RPC operations caused by network congestion [12, 77].

One promising approach to controlling tail latency is to adjust the network configuration. Data center networks expose numerous parameters that significantly influence tail latency, including congestion control thresholds, switch scheduling weights, buffer management settings, and more. Properly configured, these parameters can reshape queueing behavior to meet SLOs without requiring additional capacity. Recently, automated methods for parameter tuning have gained traction [4, 52, 81]. These methods iteratively probe possible configurations, adjusting based on observed performance to achieve operator goals. They are *model-free*: they treat the network as an opaque black box and discover good settings through search. However, this iterative trial-and-error becomes increasingly untenable as dimensionality grows. With each additional parameter, the space of potential configurations expands exponentially, causing prohibitively slow convergence and frequent exploration of poor configurations. Moreover, because these methods must physically deploy configurations to evaluate them, unconstrained exploration can make performance arbitrarily worse, directly degrading performance. This is especially true in dynamic environments where workload changes mean that the optimal settings may shift. Dynamic workloads also confound attribution: an observed change in tail latency may stem from the workload rather than the configuration, making black-box search unreliable. Because SLOs are calculated on customer-visible behavior, the tail latency for any poor configuration counts against the SLO, potentially causing operators to increase overprovisioning to compensate.

What if instead we had a performance model to help guide this parameter search? If we could *predict* a configuration's tail latency impact before deploying it—quickly and with acceptable accuracy—we could explore large parameter spaces without risking violations. Of course, unlike model-free approaches that only monitor results, such predictions require measuring workload characteristics as input, since network performance depends on the workload. Moreover, if in addition to the predictive model and the workload we had an online *controller* that used the model's predictions to guide its decisions in closed-loop , we could continuously steer the network toward SLO compliance, in spite of a dynamic environment.

Thesis: prediction-guided control enables adaptive SLO compliance.

My thesis is that *prediction-guided control provides a basis for efficient, effective, and adaptive SLO compliance in data center networks.* I support this claim in two parts: first, by constructing a model that makes fast and accurate tail latency predictions under different configurations, and second, by proposing and evaluating an online controller that uses those predictions to maintain compliance in dynamic, high-dimensional environments.

1.1 CONSTRUCTING A FAST TAIL LATENCY ESTIMATOR

For prediction-guided control to work well, how fast should tail latency estimates be? Tail latency percentiles require enough samples before measurements become meaningful, and because the metric is inherently noisy, it is commonly measured on the timescale of minutes [31, 40]. Ideally, we should be able to adapt configurations on the same timescale that we can measure them, so if stable observations emerge every few minutes, control decisions should keep pace.

Because networks have so many moving pieces, *simulating* their behavior is the most common approach to estimating the impact of configuration changes. Unfortunately, existing tools are too slow for this setting. Packet-level simulators like ns-3 trace each packet through every device and every link, faithfully reproducing the fine-grained feedback loops between network components [98], but this fidelity comes at a cost: simulating just a few seconds of traffic in a mid-sized network can take hours or days [95]. One obstacle to speeding up simulation is tight coupling. As packets flow through the network, the scheduling decisions at each switch affect the behavior of every flow traversing that switch, and therefore the scheduling decisions at every downstream switch, and—with congestion control—future flow behavior, in a cascading web of very fine-grained interaction. These interdependencies make it difficult to parallelize the simulation across cores or nodes. Recent state-of-the-art simulators like DONS and UNISON speed up simulation by 21-65× [11, 36], but still require tens of minutes to simulate even a few seconds of traffic on large topologies typical of modern data centers.

To speed up estimates, we make two observations. First, while flow-by-flow latency estimates are helpful for debugging or protocol design, they are not necessary for controlling tail latency. Since most SLOs are defined on the basis of aggregate network performance, we only need to estimate how configuration changes affect the *distribution* of flow latency. Second, if we can avoid modeling the interaction between switch queues, we can simulate each link in isolation. This decoupling would make simulations embarrassingly parallel, allowing multicore and multiserver architectures to reduce the time to make predictions. Chapter 3 describes `Parsimon`, which develops and evaluates this idea. It breaks down the network into per-link simulations, runs them independently using a simplified network model, and aggregates the results to approximate end-to-end tail latency. The key tradeoff is precision for speed: we discard per-flow estimates but retain statistical accuracy in aggregate.

`Parsimon` takes as input a workload, topology, and configuration. It prunes redundant links, simulates the remaining ones in parallel, and combines per-link delay distributions to estimate tail latency. Compared to ns-3, `Parsimon` speeds up simulations by nearly 500×

while keeping tail latency estimates within 10% of the output of ns-3. By reducing simulation time to about a minute, `Parsimon` is fast enough to embed in an online controller, which can use it to evaluate candidate configurations under current conditions, test alternatives, and select one that veers toward compliance.

1.2 BUILDING A PREDICTION-GUIDED NETWORK CONTROLLER

A fast tail latency estimator is not enough on its own. Even if `Parsimon` perfectly replicated ns-3, it would still fall far short of guaranteeing accuracy in real networks. Simulators strive for high fidelity, but necessarily omit details like OS- and application-level effects. In our own tests, `Parsimon`'s predictions align closely with ns-3, but diverge from the behavior observed on a target system with real machines. To reduce modeling error, we also trained a state-of-the-art machine-learned model, `m3`, directly on measurements from the target system, and while it outperformed `Parsimon`, it still exhibited significant error.[2] We show that when deploying `m3`'s predictions in open-loop, it fails to meet SLOs that were achievable, indicating that prediction alone is not sufficient for effective control.

Closing the loop offers a natural path forward by adapting configurations in response to real-time measurements. However, closed-loop control with approximate models introduces additional challenges. The controller needs to correct for modeling error online, learning where and how predictions diverge from reality. Adaptation must balance exploration with stability, reducing the risk of SLO violations during the search process itself. Finally, the controller needs workload characteristics as input to the performance models, requiring continuous measurement of flow-level data from the network. Addressing these challenges requires integrating model evaluation, error correction, constrained optimization, and workload monitoring.

Chapter 4 describes `Polyphony`, a prediction-guided network controller that addresses these challenges. `Polyphony` uses approximate models like `Parsimon` and `m3` not as perfect oracles, but as *prior beliefs*: it combines their predictions with real-time measurements to make decisions. At each control step, it fuses model estimates with observed outcomes to refine its belief about the system's behavior. It then selects a promising configuration using a balance of exploration and exploitation. This composite approach allows `Polyphony` to benefit from fast model evaluations while correcting for errors that emerge in deployment. Integrating continuous feedback turns an unreliable predictor into a more dependable guide.

2 `m3` is subsequent work to `Parsimon`, led by our collaborators at MIT. It generally outperforms `Parsimon` in both speed and accuracy, but requires gathering training data on the target system.

In addition to correcting for model error, `Polyphony` must remain safe and adaptive. To stay safe in the presence of inaccurate predictions, it restricts exploration to a fixed trust region centered on the best configuration seen so far. This limits the impact of bad estimates and prevents the controller from making large, risky changes. To remain adaptive, `Polyphony` monitors system behavior for signs of change. When it detects a regime shift, such as a spike in load, it resets the trust region to allow broader search. It also forgets stale observations over time, discarding outdated measurements so they do not bias future decisions. These mechanisms allow `Polyphony` to correct for prediction errors and track shifting optima as conditions change. In experiments on CloudLab, `Polyphony` converges to tight SLOs within minutes, while a recent model-free tuner fails to converge within an hour.

1.3 PUBLISHED WORK

This dissertation includes material from the following papers:

- Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. "Scalable Tail Latency Estimation for Data Center Networks." In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 685-702. 2023.
- Kevin Zhao, Chenning Li, Anton A. Zabreyko, Arash Nasr-Esfahany, Anna Goncharenko, David Dai, Sidharth Lakshmanan, Claire Li, Mohammad Alizadeh, and Thomas E. Anderson. "Prediction-Guided Control in Data Center Networks." *In submission.*

It also builds on work from:

- Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. "m3: Accurate Flow-Level Performance Estimation Using Machine Learning." In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 813-827. 2024.

1.4 SYNOPSIS AND OUTLINE

The rest of this dissertation develops and supports the thesis: that prediction-guided control can enable efficient, effective, and adaptive SLO compliance in data center networks. It does so by building a fast estimator to predict tail latency under configuration changes, and by using those predictions to guide a feedback-driven controller.

Chapter 2 reviews prior work on managing latency in data center networks. It covers topics like quality-of-service mechanisms, congestion control protocols, and more recent efforts to autotune network

configurations using measurement-driven and learning-based methods. These approaches highlight both the promise and limitations of reactive control, and motivate the need for predictive techniques.

Chapter 3 presents and evaluates `Parsimon`, a fast tail latency estimator. `Parsimon` simulates each network link independently and aggregates results to approximate end-to-end latency. This structure allows near-linear speedup in the number of cores with minimal accuracy loss compared to ns-3. The chapter demonstrates that `Parsimon` can reduce simulation time by multiple orders of magnitude while maintaining accuracy sufficient for control.

Chapter 4 presents and evaluates `Polyphony`, a prediction-guided controller that uses approximate models such as `Parsimon` or `m3` to select configurations. `Polyphony` continuously collects workload data from the network, uses models to evaluate candidate configurations, and deploys promising ones to the system. It corrects for model error by integrating real-time feedback, constrains exploration to a trust region for safety, and adapts to workload shifts by forgetting stale data and monitoring for regime changes. This chapter describes how `Polyphony` can track shifting optima, avoid SLO violations, and converge to good configurations significantly faster than model-free approaches.

Chapter 5 discusses limitations of the work and proposes directions for future research.

2

BACKGROUND

Before we describe how prediction-guided control can help maintain SLOs, we take a step back. For decades, network designers have wrestled with the question of how to deliver good performance for important traffic at scale. Their answers for data center networks include traffic classes, switch weights, congestion control, traffic engineering, and capacity planning. This chapter traces the lineage to situate our approach in a broader, decades-long effort to make networks more responsive and efficient.

2.1 CLASSICAL QUALITY OF SERVICE

QoS means deliberate traffic differentiation.

The idea that a network should treat some traffic better than others—intentionally and predictably—emerged as the Internet began to carry more than email and file transfers. Interactive applications like voice and video revealed a limitation of the original IP design: it made no distinction between packets that could wait and those that couldn't. By the mid-1980s, researchers had reframed this issue as a resource allocation problem. How should bandwidth, buffers, and schedule slots be divided so that some flows receive bounded delay or loss, while others proceed best-effort? That question launched a decades-long effort toward what became known as quality of service (QoS). For our exposition, we call out three broad phases: per-flow fairness, per-flow reservation, and class-based differentiation.

2.1.1 *Per-flow fairness*

The earliest approach to QoS was rooted in fairness: if each flow received an equal share of a link, then no single sender could crowd out the rest. This was especially important as the Internet transitioned from bulk transfers to interactive workloads, where latency spikes could disrupt entire applications.

The idea began to take shape in 1985, when John Nagle proposed *fair queueing*: a simple round-robin scheduler that interleaved packets across flows [67]. Later refinements added flexibility and precision. Weighted fair queueing (1989) allowed some flows to receive more bandwidth than others, using virtual finish times to compensate for unequal packet sizes and unequal weights [28]. By 1995, deficit round robin provided an efficient implementation of an approximate version of weighted fair queeuing, reducing per-packet cost to constant time [78].

Weights are shares of link capacity.

These designs introduced the concept of a *weight*—a numeric share of a link's capacity—which survives today in the weighted-round-robin logic of nearly every commodity switch. But the cost of fairness as originally conceived was per-flow state: core routers had to track and schedule thousands of flows in real time. Later work like Core-Stateless Fair Queueing (CSFQ) showed that approximately fair bandwidth allocations were possible without per-flow state [82], but CSFQ requires operations unavailable in commodity switching hardware and did not see wide deployment.

2.1.2 *Deterministic bounds, traffic shaping, and per-flow reservation*

As networks carried more interactive traffic, fairness was no longer enough. Fairness is inherently relative: a flow's share depends on the number and weight of competing flows, making it unsuitable for providing absolute guarantees. Some applications needed guarantees: not just a share of bandwidth, but bounds on delay and loss. This led to the idea of reserving resources along the path of each flow.

The theoretical foundations for such guarantees emerged through network calculus. In the early 1990s, Cruz introduced the language of arrival curves and service curves [25, 26], showing how traffic regulators and schedulers could be composed to yield end-to-end bounds. Parekh and Gallager proved that generalized processor sharing (GPS) provides strict rate isolation, in the sense that each flow's service depends only on its own reserved rate, not on the behavior of other flows [68]. These results assume specific conditions: flows shaped at ingress using token buckets, which regulate traffic to a burst size σ and long-term rate ρ; scheduling that approximates GPS, giving each flow a minimum rate $R \geq \rho$; and admission control ensuring the sum of reserved rates never exceeds link capacity. The key insight is that shaping at ingress prevents burst coupling. Without shaping, bursts from many flows can stack unpredictably at a bottleneck, potentially overrunning switch buffers and allowing one flow's rate to affect another's (e. g., by causing packet loss). With shaping, each hop only ever "owes" a bounded backlog to each flow, so per-hop delay bounds compose cleanly across the path. Together, these mechanisms provide worst-case bounds on delay, backlog, and jitter—true guaranteed QoS.

Shaping + minimum-rate scheduling + admission ⇒ deterministic bounds

Integrated Services (IntServ) was an architecture designed to realize these ideas in practice [18]. Formalized in RFC 1633, IntServ defined two service models. *Guaranteed service* aimed to provide strict delay and jitter bounds [43]. *Controlled load* sought to approximate the experience of an unloaded network. To use either, a flow described its traffic using a specification (`TSpec`) and optionally requested performance bounds (`RSpec`). These guarantees have rigid requirements: per-flow traffic shaping, per-flow scheduling, and admission control in every router along the path. The approach works well for flows with pre-

dictable traffic envelopes, like voice or video, where the application can describe its needs in advance.

Supporting IntServ required routers to participate in signaling.[1] The RSVP protocol, standardized in 1997, handled this task [19]. The immediate motivation was to prepare the Internet for streaming video: if applications could reserve bandwidth, ISPs could provision accordingly and offer quality guarantees. Receivers initiated reservations, routers along the path installed soft state[2] recording the reservation, and refresh messages kept that state alive. Each hop could admit or reject requests, depending on available capacity.

In controlled environments, the model worked. But in the open Internet, per-flow reservation faced a deployment paradox. The system only provided value with end-to-end support, yet no single ISP could guarantee performance across network boundaries. Without broad adoption, incremental deployment offered little benefit, and without clear benefit, operators had no incentive to deploy. What's more, deployment had large technical costs. Core routers had to maintain per-flow state, run admission control, and schedule traffic accordingly—costs that grew with link speeds and flow counts. The reliance on soft state made the system brittle: when a router failed or rebooted, all reservations traversing it expired, requiring flows to re-request admission, which could be denied if capacity had since been allocated elsewhere. These operational challenges might have been worth solving if the problem was urgent, but rapid growth in backbone bandwidth reduced that urgency. Provisioning more capacity was simpler and often cheaper than coordinating a technical solution across competing organizations and different geographic regions. Meanwhile, adaptive bitrate (ABR) encoding emerged as a more practical solution for streaming video, applying the end-to-end principle: if a reservation fails or a route changes, ABR simply adjusts the video quality, whereas RSVP would drop the stream entirely. IntServ was implemented at network edges, where flow counts were modest, but was abandoned in the core. The idea of guaranteed service remained compelling, but per-flow reservation did not see wide adoption on the public Internet, even though the theory was sound and niche environments could adopt pieces of it. This experience motivated a pragmatic alternative: rather than track every flow, aggregate traffic into a small number of classes that could be managed with simpler mechanisms, particularly within networks managed by a single organization.

2.1.3 *Class-based QoS*

This shift toward aggregation took hold in the mid-1990s with class-based queuing (CBQ) [35]. Rather than scheduling individual flows,

1 Exchanging control messages to coordinate resource allocation.
2 Temporary state that expires unless periodically refreshed.

CBQ allocated bandwidth among a hierarchy of classes. Each class could specify a minimum share, a maximum cap, and a policy for how sibling classes should share surplus. The design was flexible, and more importantly, tractable: a small number of classes could be scheduled efficiently on commodity hardware.

DiffServ extended these ideas to Internet scale [16]. It moved classification to the network edge, where ingress routers mark each packet with a Differentiated Services Code Point (DSCP). Core routers then applied a per-hop behavior (PHB) based on that tag, typically by assigning the packet to a queue per DSCP, assigned a fixed weight or priority. This division of labor kept the fast path simple: core routers needed no flow state, no signaling, and no negotiation.

By the early 2000s, DiffServ was widely deployed. ISPs offered class-aware VPNs, enterprise networks enforced QoS with DSCP and VLAN tags, and switch vendors exposed just enough knobs—queues, weights, priorities—to support basic differentiation. The resulting model was coarse but durable. It abandoned per-flow guarantees but kept enough structure to manage performance across competing traffic classes.

Modern data centers inherited this architecture. The prevalence of short, bursty flows makes advance per-flow reservation impractical (although some have proposed it [69]), and the decisive factors remain operational: simplicity of per-class mechanisms, ease of provisioning, and compatibility with existing switch hardware. Accordingly, hyperscale operators favor a small number of priority classes with shaping and policing at the edges, augmented by congestion signaling, transport tuning, and traffic engineering (discussed in §2.2).

Deterministic bounds require pessimistic rate limits.

With sufficient traffic shaping, class-based mechanisms can provide deterministic latency bounds. QJump demonstrates this by rate-limiting each host so that aggregate traffic at a given priority level cannot overflow switch queues, even in the worst case [42]. But the rate-latency tradeoff is steep: bounding delay requires bounding queue occupancy, which in turn requires limiting the arrival rate. Worst-case analysis assumes all hosts may transmit simultaneously, so the highest-priority class is typically restricted to a few percent of link capacity. This suffices for small control-plane messages or heartbeats, but cannot extend guarantees to the bulk of data center traffic.

This dissertation operates in the same setting of class-level knobs, but rather than pursuing deterministic bounds for a small fraction of traffic, we aim to maintain tail-latency SLOs for almost all traffic, despite temporary overload and across a mixture of flow sizes.

2.1.4 *Legacy and limitations*

Many ideas from classical QoS—weights, classes, and traffic shaping—are alive and well to this day. Today's switches support 8 to 16 output queues per port, configurable weights for round-robin or deficit

scheduling, strict-priority flags, token-bucket policers, and DSCP-based classification. These are knobs that engineers reach for when they need to manage traffic.

QoS primitives: traffic classes, separate queues, queue weights

But some of the assumptions that underpin classical QoS no longer hold in today's networks. Classical QoS imagined a world where flows were long-lived, where traffic patterns changed slowly, and where human operators could hand-tune parameters based on long-term measurements. Much of the work optimized mean performance. Today's data centers operate under different conditions. Services are composed of thousands of microservices issuing billions of short-lived RPCs. Traffic can shift over minutes or hours, not just days or weeks, and performance targets focus on tail latency, not the mean.

Meanwhile, the control surface has grown. In addition to queue weights and class tags, operators must tune ECN thresholds, buffer partitions, PFC settings, congestion-control parameters—interdependent variables that interact in ways that defy easy characterization.

This dissertation builds on this machinery. We make use of core concepts, like queues, weights, and traffic classes, but seek to automate their configuration in an adaptive way.

2.2 CAPACITY PLANNING AND TRAFFIC ENGINEERING

An alternative to differentiating traffic is to provision enough capacity that performance problems rarely arise. If links run at low average utilization—often on the order of a few tens of percent of capacity—then traffic bursts are less likely to cause persistent queueing. When it works, this approach is much simpler than QoS mechanisms which may require per-flow state or signaling protocols. The strategy is effective but expensive, and extra capacity benefits all traffic equally, whether or not it is latency-sensitive. However, static provisioning does not adapt to shifts in demand. Even a well-provisioned network can develop hotspots when traffic concentrates on some links while others remain underutilized.

Overprovisioning trades cost for simplicity.

Traffic engineering (TE) addresses this by redistributing load across available paths. Rather than adding more links, TE adjusts routing to spread load across available paths and avoid hotspots. In wide-area networks, Multiprotocol Label Switching (MPLS) allowed operators to establish explicit paths and steer traffic away from congested links [62]. Data centers have adopted simpler mechanisms. Equal-cost multipath (ECMP) routing hashes flows across parallel paths, providing coarse load balancing without centralized coordination [49]. ECMP is simple and widely deployed, but it is static in that it does not react when demand shifts or when a few large flows collide on the same path.

Traffic engineering optimizes routing.

Adaptive traffic engineering systems emerged to address these limitations. Hedera detected elephant flows and rerouted them to less-loaded paths [2]. CONGA performed fine-grained, flowlet-level

load balancing using congestion feedback from switches [5]. At the wide-area scale, Google's B4 and Microsoft's SWAN used centralized controllers to recompute routes every few minutes based on measured demand, achieving high utilization without persistent congestion [48, 51]. These systems demonstrated that continuous measurement and adjustment can extract substantially more performance from fixed infrastructure.

Recent work has pushed this idea further by making the physical topology itself reconfigurable. Google's Jupiter fabric uses optical circuit switches to provision bandwidth dynamically between aggregation blocks [70]. When demand between two blocks persistently increases, the system can reconfigure optical circuits over relatively coarse timescales to allocate more bandwidth between them. This approach blurs the line between capacity planning and traffic engineering: rather than provisioning capacity once and routing around it, Jupiter periodically provisions capacity in response to demand.

All of these systems follow a common structure: measure traffic, compute a better configuration, apply it, and repeat. This closed-loop, feedback-driven approach is also central to the work in this dissertation. But traffic engineering differs in that it primarily adjusts routes, deciding which paths traffic should take through the network, and its objectives typically involve aggregate metrics like minimizing maximum link utilization or achieving max-min fairness among flow groups. By contrast, this dissertation considers a much broader control surface, and we target per-class tail latency SLOs, which depend on more factors than the distribution of load across links. Traffic engineering and capacity planning remain essential tools for network operators, but even a well-provisioned, well-routed network can still experience tail latency violations.

2.3 CONGESTION CONTROL

Congestion control manages how quickly flows send packets.

While traffic engineering balances the utilization of links in the network, modern data center congestion control aims to limit tail latency and, particularly, packet loss. Most schemes operate at the transport layer, adjusting a sender's rate in response to signs of congestion. In data center networks (DCNs), where traffic is bursty and round-trip times are short, congestion control plays a central role in shaping queueing behavior, and, by extension, latency and tail latency. Accordingly, designing better congestion control algorithms was the first approach to improving SLO compliance in data center networks.

Originally, congestion control was desigend to provide good performance regardless of offered load. When the Internet began carrying significant traffic in the 1980s, overly-aggressive senders could cause cascading failures as routers dropped packets that then had to be resent, consuming even more resources [24]. Early TCP implementa-

tions addressed this by probing for available bandwidth, increasing the sending rate until packets were dropped, then backing off if they detect overuse. This approach keeps links utilized without overwhelming them, but is optimized for throughput and fairness, not latency. Variants like TCP Reno and CUBIC refine these dynamics but retain the same goal of avoiding poor throughput and efficiently sharing capacity [44].

Data centers operate under different constraints. Rather than long-lived bulk transfers over wide-area paths, they carry billions of short RPCs over low-latency fabrics. In this setting, the role of congestion control shifts from preventing poor throughput to managing a tradeoff between latency and throughput: accept some bandwidth loss in exchange for more reliable low latency and low queueing. DCTCP [6] is an example. It uses ECN marks as an early congestion signal, scaling the congestion window based on the fraction of marked packets rather than waiting for packet loss to singal congestion. By reacting before queues completely fill, DCTCP reduces average queueing delay, trading a small amount of throughput for reduced latency, tail latency, and packet loss. This principle—using congestion control not just to keep links full, but to constrain queues for latency-sensitive traffic—has informed later congestion control algorithms like DCQCN [97], HPCC [58], and Swift [54].

These refinements reduce average and tail queueing delay, but unlike QoS solutions they do not guarantee bounds on delay. Most of these schemes share the same basic structure: a packet carries congestion information from the switch to the receiver, which then returns the signal to the sender. Only after this round-trip can the sender adjust its rate. The result is a feedback delay with a lower bound of the RTT [41]. This delay becomes more costly as link speeds increase. At 100 to 400 Gbps, the bandwidth-delay product—the amount of data in flight before any adjustment takes effect—reaches hundreds of kilobytes inside a data center.[3] Depending on the host policy, a sender may fill a queue with a full BDP of traffic before receiving any signal. This could be enough to push queueing delay well beyond the thresholds needed to meet tail latency SLOs for other traffic sharing the same queue.

Round-trip time is a lower bound for feedback delay.

A problem is that most congestion control algorithms are designed under the assumption that flows last many round trips, giving them time to probe for available bandwidth, react to congestion signals, and converge to fair allocations. However, in modern data center settings with small messages and high link speeds, the vast majority of flows are short-lived, with many of them completing within a single round trip. These flows never reach the steady-state behavior that congestion control was designed to regulate. Instead, their behavior is controlled

3 Assuming a 12μs RTT.

by startup parameters that determine how quickly they can inject packets before any feedback is received.

One such parameter is the *initial congestion window*, defined as the number of packets a sender may transmit before receiving an acknowledgment. Larger initial windows can improve throughput for long flows, but they increase the risk of queue buildup, especially when many flows start at once. Smaller windows help keep queues short, but can unnecessarily stall throughput if they are too conservative.

To keep tail latency in check, operators often run links well below capacity, sacrificing efficiency to preserve headroom. Some research proposals aim to overcome the RTT lower bound on feedback delay by pushing control into the network. For example, Backpressure Flow Control (BFC) embeds per-hop, per-flow rate signals directly into packets, allowing switches to throttle senders in microseconds [41]. Simulations and hardware prototypes show dramatic reductions in tail latency without loss of throughput. But such schemes require specialized hardware and are not widely deployed. In production, congestion control predominantly relies on end-to-end feedback that requires at least one round trip: DCTCP, DCQCN [97], HPCC [58], and others all operate on RTT timescales. This means that flows completing in less than one RTT remain effectively uncontrolled by these mechanisms.

This dissertation treats congestion-control parameters as part of the control surface. While future work could explore designing congestion control algorithms specifically to meet per-class tail latency SLOs, here we do not alter the congestion control algorithm itself; instead, we tune its parameters and configuration. This choice allows us to work with existing deployed protocols. In particular, we focus on DCTCP's ECN marking threshold—the queue length K at which switches begin setting ECN bits—and the initial congestion window. These shape how early congestion is signaled and how much data flows before feedback can take effect. By adjusting these parameters dynamically, our controller can pick a latency-throughput tradeoff that is appropriate for a particular set of SLOs. This represents a shift from reactive, per-flow control to proactive, network-wide optimization: rather than waiting for each flow to discover congestion and adjust, we configure the system in advance to reduce the likelihood of tail latency violations across the entire workload. Although we focus on DCTCP, other congestion control mechanisms have similar knobs that can potentially be adjusted in similar ways. We leave this for future work.

Tunable: ECN thresholds, initial congestion window

2.4 IN-NETWORK FLOW SCHEDULING TRANSPORTS

Congestion control adjusts how quickly flows send packets into the network. But when many flows converge on the same queue, control-

Packet order affects latency, not just send rate.

ling the send rate may not be enough to ensure low latency. The order in which packets are served also matters: a small, latency-sensitive request may sit behind a large transfer and incur queueing delay it could otherwise avoid. To address this, researchers asked: what if the switch could prioritize more urgent packets?

This question led to a class of in-network schedulers that approximate ideal service orders—policies that, in theory, minimize missed deadlines or average flow completion time. For example, earliest deadline first (EDF) [60] prioritizes the most time-sensitive packets, while shortest remaining processing time (SRPT) [76] favors short flows that can complete quickly. Rather than throttle flows, these schemes decide which packet moves next. They act at microsecond timescales, using per-packet tags and simple switch logic to emulate sophisticated scheduling policies.

A series of proposals explored this idea. D3 admitted flows in deadline order [87]; PDQ paused low-urgency flows to emulate earliest deadline first [47]; pFabric prioritized packets with fewer bytes remaining [8]; and NDP replaced work-conserving queues with pull credits [45]. These designs showed how much latency could be reduced if the network prioritized the "right" packet. In simulations and rack-scale tests, these systems significantly improved flow completion times and reduced deadline misses compared to congestion control alone. But they were not SLO-aware: performance gains were impressive, yet tied to workload characteristics and internal tuning parameters, not driven by explicit service objectives.

Despite their promise, none saw wide deployment. Each design required the fabric to process per-packet tags at line rate, often with custom headers, parsing logic, and credit handling. Real-world adoption would also demand updates to NIC firmware, host software, debugging tools, and middleboxes that expect FIFO behavior. These changes posed significant operational barriers. Recent work on programmable scheduling primitives like PIFO has shown that these policies can be implemented more flexibly [80], but adoption remains limited in commodity hardware.

Homa adopts a different strategy. It approximates SRPT by having senders assign packet priorities based on message size, relying only on commodity priority queues already present in switches [66]. Homa also uses packet spraying to balance load across paths, which can cause out-of-order delivery; handling this efficiently requires NIC support for packet reordering. These choices avoid switch modifications, but deployment still requires replacing TCP with a new transport API, upgrading NICs, integrating Homa into RPC frameworks, and building equivalent tooling—barriers that have so far limited its adoption.

This dissertation proposes an orthogonal approach. It does not schedule individual packets, nor does it require changes to switch logic or packet formats. Instead, it retunes previously static parameters that

shape queueing and congestion: ECN thresholds, initial congestion windows, and queue weights. These inputs already exist in commodity hardware; we propose adjusting them dynamically to meet SLOs across traffic classes. The two approaches are complementary: in-network schedulers decide on good packet order, and a prediction-guided controller tunes queueing and congestion behavior.

2.5 ADMISSION AND OVERLOAD CONTROL

When pacing and prioritization aren't enough, requests must be dropped.

Congestion control reacts quickly to transient bursts, adjusting a sender's rate on microsecond timescales. In-network schedulers go further, reordering packets to better match application urgency. But when sustained load exceeds capacity over long timescales, even precise pacing and prioritization fall short of providing good performance. At that point, the system must decide not just how fast to send or which packet to serve next, but whether to serve a request at all.

These mechanisms act at the boundary of the system, determining whether to admit, delay, downgrade, or drop new requests [23, 89, 94, 96]. Their goal is to protect service-level objectives for latency-sensitive or high-priority traffic by preventing systemic overload. SLOs, by definition, apply only to traffic that the system admits; protecting these guarantees sometimes requires rejecting or delaying even high-priority work. Some approaches are sender-driven, using probabilistic admission or token buckets [94]. Others target server overload rather than network congestion, using receiver-driven admission control based on server queueing delay or load [23]. Most are reactive, designed to intervene only once overload is already underway, and rely on relatively static policies to enforce admission decisions.

SLOs are only met for admitted traffic.

This dissertation does not displace admission control, but builds around it, tuning the system before rejection becomes necessary. Like admission control, our approach does not completely prevent long queues. But by improving how the system uses available headroom, it can delay the need to reject traffic and reduce the severity of overload when it occurs. We do this by adjusting network parameters based on workload conditions. When overload does occur, our controller leaves admission control to operate as usual. But in principle, those same decisions—whether to drop, defer, or prioritize—could be incorporated into the predictive loop as tunable variables, rather than fixed rules. We leave this for future work.

2.6 MODEL-FREE AUTOTUNERS

Earlier sections illustrate that reducing tail latency in data centers depends on many interacting layers. Flows compete for queueing priority, queues grow and shrink based on congestion control feedback, and system-wide overload triggers admission control. In the midst

of all of these dynamics is a set of parameters: queue weights, buffer allocations, initial congestion windows, switch marking thresholds, traffic pacing, and other congestion and traffic control settings. These settings are consequential, shaping queue growth, sending rates, and flow priority at the switch. Though they are already present on nearly every switch and host, they are often fixed at deployment and rarely revisited. When workloads shift, due to changing services, diurnal patterns, or infrastructure updates, static configurations can lead to queues forming in the wrong places and pressure to expand capacity even when much of the bandwidth is left unused. Retuning them is difficult: the parameters interact in complex ways, their effects are nonlinear, and the system is noisy. Manual adjustment may be slow and error-prone.

This challenge has led a number of researchers to propose automated tuning through observation. *Model-free autotuners* take this approach by treating the system as a black-box: they try a configuration, measure its effect, and update accordingly. The method is general: it requires no traffic model and no domain-specific logic. Bayesian optimization formalizes black-box search as a probabilistic inference problem. It builds a model of the objective function (often a Gaussian process) and uses it to select the next configuration. Each choice balances two goals: exploring uncertain settings to gather information, and exploiting known good settings to improve performance. This tradeoff makes Bayesian optimization well-suited to problems where evaluations are costly and gradients unavailable.

Try, measure, update: learning by probing.

This approach works well when each trial can be run with limited scope and manageable risk. Systems like CherryPick [4] and Metis [59] apply Bayesian optimization to tune single-service or host-level settings, like virtual machine types, kernel flags, and runtime parameters. A poor choice may waste compute or degrade latency, but the impact is contained and visible only to that service. In data center networks, the cost is steeper. Each trial changes global queueing behavior for live traffic, affecting tail latency across many services. The effect propagates throughout the network and can breach SLOs system-wide before the tuner converges.

Another emerging model-free approach is reinforcement learning. Systems like SelfTune [52] and OPPerTune [81] adjust parameters in rounds: each round selects a configuration, runs it under load, and records a scalar reward. These methods work well when each round yields clear, stable feedback—typically in systems with long timescales and strong gradient signals. In data center networks, the conditions are different. Tail latency varies as a function of workload even under fixed settings. Small gains in performance are hard to distinguish from noise, and without strong gradients and informative rounds, learning slows or stalls.

Toward safer tuning: predict before probing.

Bayesian optimization and reinforcement learning tuners differ in mechanics but share a limitation: they learn by probing the live system. Each exploratory step risks performance regressions that violate SLOs, and exploration must be repeated when the workload shifts. This raises a natural question: can we predict the effect of a configuration before applying it? The next section explores this line of work.

2.7 EMERGING FAST PREDICTORS

Effective control depends on foresight. To select a configuration that meets tail latency SLOs, a system must be able to predict how that choice will affect tail latency. The challenge is not that predictions must be perfect, but that they must be fast, reliable, and accurate enough to guide decision-making. Two traditional methods offer a foundation for prediction: analytical modeling and system identification. Each takes a different view of what it means to understand a system, but both run into limits in the context of data center networks.

Analytical modeling works best in domains governed by physical laws. Engineers can model the thermal behavior of a chip or the torque of a motor because the underlying processes obey well-understood equations. Computer networks do not enjoy the same luxury. While queueing theory provides formal models of shared resources, its assumptions, like stationary Poisson arrivals, rarely match real-world conditions. Data center traffic tends to be bursty and dominated by short flows, and congestion control introduces feedback loops that couple behavior across switches and hosts.

System identification offers a more empirical path. It treats the network as an unknown dynamical system and tries to infer a model from observed behavior. In theory, this approach can capture arbitrary complexity without requiring an explicit understanding of how the system works. In practice, however, it demands broad and representative data: the system must be pushed through enough different states to reveal its structure. This is true in robotics, where actuators can be exercised repeatedly over a known operating range. But in data centers, experimentation carries risk. A single poorly-tuned switch weight can spike tail latency for critical services. The configuration space spans many interdependent knobs, and the workload may shift frequently. Collecting a diverse training set without violating SLOs is, in effect, intractable.

When real-world experimentation is unsafe or impractical, *simulation* has long been the default tool for exploring system behavior under new conditions. It offers a controlled way to test new protocols, compare configurations, and explore system behavior at scale. But detailed simulators have historically been slow: packet-level simulators like ns-3 [98] offer high fidelity but take hours or days to simulate a few seconds of a large network [95]. Recently, a wave of work has focused on

accelerating these methods. Systems like DONS [36] and UNISON [11] parallelize event processing to improve throughput without losing detail. These designs achieve significant speedups, but still require tens of minutes to simulate large networks, which is still too slow to guide parameter tuning in real time. Other systems, like DeepQueueNet [91] and MimicNet [93], use machine learning to approximate network dynamics directly. DeepQueueNet can forecast per-packet delays from packet traces, but does not model congestion control, limiting its ability to predict the impact of tuning. MimicNet retains full transport fidelity but is limited to symmetric FatTree topologies in failure-free scenarios; its core speedup relies on replicating a single observable cluster, which is not applicable to heterogeneous or irregular networks. Such irregular networks are common in practice simply because of the need to upgrade while the network is being used [70]. These approaches demonstrate that fast prediction is feasible, but each comes with tradeoffs in scope and generality.

The question is whether a predictor can strike a better balance. Can it estimate tail latency across arbitrary workloads and topologies, fast enough to guide configuration in real time? The next chapter describes a simulator we have designed for this purpose.

3 SCALABLE TAIL LATENCY ESTIMATION

3.1 INTRODUCTION

This chapter introduces a fast approximate simulator designed to guide parameter and algorithm selection in data center networks. Our goal is to develop techniques that apply to large scale networks with arbitrary workloads and topologies. To speed up simulation, we ask whether network simulations can be made embarrassingly parallel.

Network simulations are hard to parallelize because packet interactions through networks of queues create global dependencies that resist decomposition. As we discussed in §2.7, even state-of-the-art parallelization techniques do not speed up simulations enough to guide real-time control. A key observation for this work is that we could achieve high degrees of parallelism if we could somehow disentangle the interactions between switch queues, allowing us to study the behavior of the traffic on each link in isolation. Of course, switch queues are not in reality completely disentangled. The packets for any particular flow experience a very specific set of conditions at each switch, and those conditions are affected by the presence of upstream bottlenecks which can smooth packet arrivals for competing flows at downstream switches. The congestion response for a flow depends on the combination of conditions at every switch along the path.

However, large scale data center networks are typically managed with the goal of delivering consistent high performance to applications. While congestion events do occur, they are often chaotic rather than persistent, popping up and then disappearing in different spots due to the inherent burstiness and flow size distribution of applications, rather than due to some long-term mismatch between demand and capacity in some portion of the network [92]. Further, we are often interested in *aggregate* behavior, such as the frequency of poor flow performance, rather than understanding the behavior or performance of each individual packet or flow.

To model aggregate behavior, our hypothesis is that we can approximate the distribution of end-to-end flow performance for a particular workload running on a large scale network by modeling the frequency and magnitude of local congestion events at each link along individual paths. A long flow will of course experience multiple congestion events during its lifetime, but most of these will occur at different points along the path *at different times*. Modeling the effect of simultaneous congestion events, and the response of the congestion algorithm to multiple simultaneous bottlenecks, is second order.

Our hypothesis is related to the concept of product-form solutions in queuing theory. For certain classes of queueing networks (e.g., Jackson [50] and BCMP networks [15]), the equilibrium distribution of queue lengths can be written in product form, i.e., the state of an individual queue is only dependent on the traffic it receives and not on the state of the rest of the network. These results generally require specific assumptions about job arrival processes (e.g., Poisson), service-time distributions (e.g., Exponential), and queueing/routing disciplines (e.g., FIFO or processor-sharing queues), and there has been much theoretical work on identifying classes of queueing networks that admit product-form solutions [53]. Although data center networks do not strictly conform to these conditions and the dynamics of each individual queue can be quite complex (e.g., due to congestion control), our hypothesis is that product-form solutions are approximately true in most realistic settings, and therefore we can analyze individual queues in isolation and combine the results to approximate end-to-end network behavior.

We built `Parsimon` to directly test this hypothesis. First, we deconstruct the network topology into a large number of simple and fast simulations where each can be run entirely in parallel by a single hyperthread. Each simulation aims to collect the distribution of delays that flows of a particular size would experience through a single link, assuming that the rest of the network is benign. We then combine these simulated delay distributions to produce predictions of the end-to-end delay distribution, again for flows of a given size. At each step, we make conservative assumptions for how delays should be computed and combined. In many settings, researchers and operators are interested in keeping tail behavior well-managed, making a conservative assumption more appropriate than an optimistic one. Finally, `Parsimon` clusters links with common traffic characteristics, eliminating much of the overhead of simulating parallel links in the core of the network as well as edge links used by replicated or parallel applications, further improving simulation performance.

Because validation against detailed packet-level simulation at scale is so expensive, we focus our study on a single widely used transport protocol, DCTCP [6], with FIFO queues with ECN packet marking at each switch [73]. We also focus on queue dynamics rather than packet loss; most data center networks are provisioned and engineered for extremely low packet loss [74, 79]. We note that these assumptions are not fundamental to our approach. We show `Parsimon` generalizes to two other transport protocols, DCQCN [97] and the delay-based TIMELY [64]. Validation of other transport protocols [7, 54, 58, 66], switch queueing disciplines [3, 28, 41, 66], and packet loss remains future work. We note that modern data center transport layer protocols are adept at quickly adapting to the presence and absence of

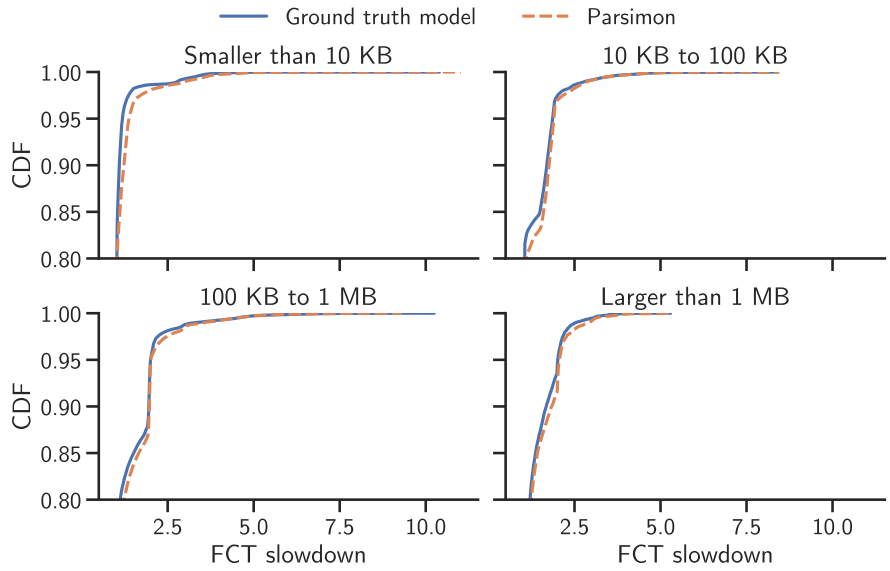


Figure 3.1: CDF of ns-3 versus `Parsimon` for flow completion time (FCT) slowdown across multiple flow size ranges, zoomed into the tail. While ns-3 took nearly 11 hours to produce these results, `Parsimon` took one minute and 19 seconds, end-to-end. Results were taken on a 6,144-host topology with an industry traffic matrix, 2-to-1 oversubscription, and bursty traffic.

congestion, and so we caution our results may not extend to older transport protocols where convergence time is a large factor.

Parsimon speeds up simulations by reasoning about links independently, which enables massive parallelization, but at a cost in accuracy. As we will see in §3.3.6, anything that creates standing congestion in the core and at the edge, or when cross traffic is correlated across multiple hops, will result in less accurate estimates. While our methods are designed to favor overestimating rather than underestimating tail latencies, this property is only evaluated experimentally (§3.5). In general there is no formal guarantee, since factors like congestion control can in theory behave in arbitrary ways that render less appropriate the approximation of considering links independently. We assume that we can simulate for long enough for the network to reach equilibrium; studies of short term transient behavior should not use our approach. We do not provide predictions at the level of an individual flow, but we are able to show that `Parsimon` is accurate for sub-classes of traffic for mixed workloads. We do not attempt to model end host scheduling delay of packet processing, even though that may have a large impact on network performance [54, 57]; we leave addressing that to future work.

To assess accuracy, we compare distributions of flow completion time (FCT) slowdown, defined as the observed FCT divided by the best achievable FCT on an unloaded network, and we say a flow is complete

when all of its bytes have been delivered to its destination. Fig. 3.1 shows a sample of our results for the 6,144 host network mentioned above, running a published industry traffic matrix [74] and flow size distribution [66], and with standard settings for burstiness and over-provisioning. We describe the details of this and other experiments later in the chapter. Depicted are FCT slowdown distributions binned by flow size. While ns-3 took nearly 11 hours on this configuration, `Parsimon` was able to match flow-size specific performance of ns-3 in 79 seconds (a 492 times speedup) on a single 32-way multicore server with an error of 9% at the 99th percentile. Given a small cluster of simulation servers, we estimate a completion time of 21 seconds using our approach.

In our evaluation, we scan the parameter space to identify circumstances where our approximations are less accurate. Link clustering improves performance but hurts accuracy somewhat; this tradeoff can be avoided by using more simulation cores. Without clustering, accuracy suffers when there is high utilization of links in the core (above 50%), there are high levels of oversubscription, and a large fraction of network traffic is due to flows that finish within a single round trip. Generally, a combination of factors is required for poor accuracy. In 85% of the configurations we test, the error relative to ns-3 is under 10%.

`Parsimon` source code and evaluation scripts are publicly available at `https://github.com/netiken`.

3.2 PARSIMON OVERVIEW

This chapter describes a set of methods to quickly and scalably estimate distributions of flow performance in data center networks. These techniques are implemented in a prototype called `Parsimon`, designed to provide the following:

- Fast, scalable estimates. We aim to supply estimates two to three orders of magnitude faster than full-fidelity simulation. Given enough cores, execution time should remain bounded regardless of network size.
- Tight latency bounds, including tail performance. Our approximations bias slightly towards overestimation, but still provide close estimates even for the 95th or 99th percentile of the distribution for a given flow length.
- Minimal restrictions on topology and workload. Our methods are largely independent of both topology and workload, although some combinations of topology and workload will have lower accuracy.

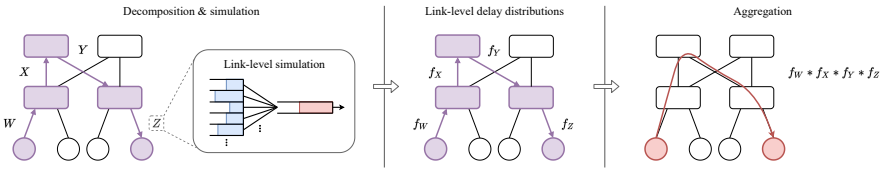


Figure 3.2: Overview of `Parsimon`. First, for any path, `Parsimon` estimates the contribution of each component link to delays in flow completion times, represented as a probability distribution. `Parsimon` then combines delays along the path using Monte Carlo simulation (see §3.3). Further, for added performance, link-level simulations are optimized and redundant simulations (due to e. g.ECMP or symmetries in workload patterns) are pruned (see §3.4).

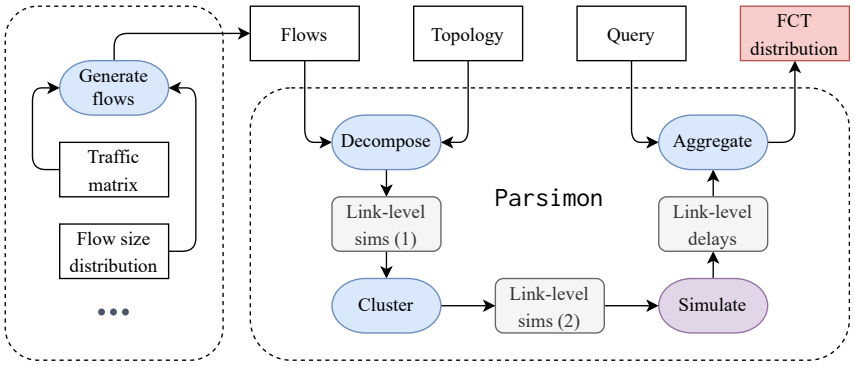


Figure 3.3: An illustration of `Parsimon`'s workflow. All inputs and outputs are shown in the top row. Rectangular boxes are inputs and outputs, rounded boxes are intermediate artifacts, and ovals are `Parsimon`'s actions.

Fig. 3.2 illustrates the intuition behind its core method, and Fig. 3.3 depicts its workflow. The user supplies 1) a description of the topology, as a set of nodes and links, and 2) the workload, as a set of flows and routes. In our implementation, we generate the flow list by sampling from the traffic matrix and the flow size distribution, with inter-arrival times determined by a burstiness parameter. Once inputs are supplied, `Parsimon` proceeds in several steps:

DECOMPOSITION. To start, flows are assigned to each link they traverse, e.g., for a fat tree using ECMP. Then, for each link l, `Parsimon` generates a custom backend simulation with a topology selected to determine—as accurately as possible—the *contribution of* l to the end-to-end flow completion times (FCTs) of the flows passing through it. Each of these backend simulations can run in parallel.

CLUSTERING. Depending on the size of the topology, there may be tens or hundreds of thousands (or more) of link-level simulations to perform. Fortunately, data center topologies exhibit notable symmetries, and industry has reported that the same is true for many

of their workloads [74]. `Parsimon` can optionally cluster links with similar workloads together. Only one representative from each cluster need be simulated; the rest of the link-level simulations are pruned. Clustering is discussed in more detail in §3.4.2.

SIMULATION. The next step is to simulate all cluster representatives in parallel. The decomposition step resulted in a topology and a workload for each link-level simulation, and we can use any simulation backend. Our prototype supports two: ns-3 and a custom high-performance link-level simulator (§3.4.1). This allows us to directly validate our link-level simulator against ns-3. However, other efficient models, such as fluid flow [63] or machine learned models could be used here instead, for different tradeoffs of performance and accuracy. Each link-level simulation produces a distribution of the delay contributed by that link to the flow completion time (FCT), bucketed by flow size. Note this is not the link's propagation delay—we calculate that contribution directly from the topology. These distributions—described in the next section (§3.3)—are organized according to the original input topology, as depicted in Fig. 3.2. Recall that only one representative from each cluster is simulated; every other link is populated with the distributions of its cluster representative.

AGGREGATION. The last step is to aggregate the link-level results into estimates for entire paths through the network. These estimates are also represented as delay distributions. Conceptually, `Parsimon` obtains a delay distribution for a path by convolving together the appropriate distributions from each of the path's component links. Since there are multiple distributions per link and potentially many paths through the network, we do not compute convolutions up-front. Instead, convolution is done on-demand via Monte Carlo sampling; a by-product is that we can efficiently produce estimates for individual source-destination pairs, virtual networks, or classes of service (§3.5.5). To make a single point prediction for a flow taking some path through the network, `Parsimon` uses the flow size to find the appropriate distribution for each link, samples a value from each of them, and combines them together. This process is repeated for each flow.

At a bird's-eye view, `Parsimon`'s method is simple: to accelerate FCT estimates, we estimate the effect of each link independently and in parallel. Then to make predictions about the whole network, we combine the results. However in our experience, the accuracy of the method hinges tightly on the quality of the link-level estimates and subsequent aggregation. For example, when generating the backend simulations, we have observed that failure to adequately capture pertinent features of the network severely degrades the quality of `Parsimon`'s estimates. Similarly, link-level results must be processed

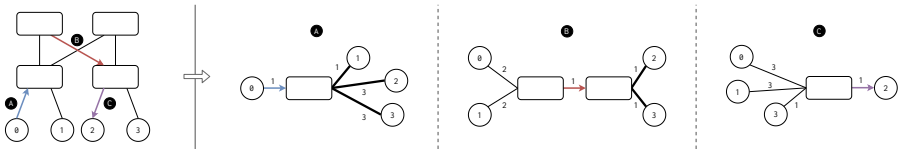


Figure 3.4: An illustration of how Parsimon generates link-level topologies. Simulations are unidirectional, and a different topology is used for (A) first-hop links, (B) switch-to-switch links, and (C) last-hop links. For illustration purposes, each link in the original topology has a propagation delay of one. To the left is the original topology; to the right are the corresponding link-level topologies, with new propagation delays annotated. Bold lines denote links whose bandwidths have been artificially increased during topology generation.

and aggregated with care to preserve accuracy across all flow sizes. §3.3 describes these techniques in detail.

3.3 KEY METHODS: DECOMPOSE AND AGGREGATE

Together, the methods for decomposition and aggregation are what enables Parsimon's scaling, and while we later engage additional techniques for further speed-up, they are a byproduct of—and not independent from—these more essential methods. Decisions made during this step are also the central determiners of accuracy. This section describes these processes in detail: how link-level topologies are generated, how the link-level data are post-processed and stored, and finally how they are aggregated to produce end-to-end estimates.

3.3.1 *Generating link-level workloads*

To start, Parsimon associates each link with the flows passing through it. Since links are bidirectional, there are two sets of flows—and consequently two link-level simulations—per link. Parsimon populates links with flows using flows' routes. Then for each link and in each direction, the associated flows constitute the input workload to the link-level simulation. The sizes and arrival times of the flows pass though unmodified.

3.3.2 *Generating link-level topologies*

Once the link-level workloads are in place, we generate the link-level topologies. In this step, we think of each link as contributing some amount of delay to end-to-end FCTs. Any given flow will accrue these delays at each hop, depending on—for example—how much bandwidth is available and how much queueing is present. Highly-loaded

links are expected to contribute more delay, while rarely utilized links will contribute relatively little.

For each link and in each direction, we generate a topology and perform a simulation using just the flows traversing that link. Once the simulation is finished, the delay caused by the link for a given flow is computed by taking the observed FCT and removing the ideal FCT for that flow size. (For a flow of size s traversing a link of speed C and propagation delay l, the ideal FCT is $s/C + l$.) This intuitively captures all delays incurred due to queueing, congestion control, bandwidth sharing, and so on at the target link.

In generating a per-link topology, our goal is to isolate and measure the expected delay contribution of the target link. A simple but inefficient strategy would be to use the original topology, but with only the traffic traversing the target link, without any cross traffic. This would be relatively accurate at measuring the delay contributed by the target link, albeit a bit conservative. Upstream cross traffic congestion will slightly smooth out downstream congestion at the target link, and so removing cross traffic would make the queue distribution at the target link slightly worse than in reality.

Although relatively accurate and parallelizable, simulating every link on the original network topology would still be inefficient, as packet-level simulation cost is roughly proportional to the number of packets simulated times the number of hops each packet takes through the network. Because we run the link simulation separately in each direction on every packet that passes through that link, this would inflate the aggregate computational cost of the simulation by a multiplicative factor of roughly half the average network path length—a significant factor for large-scale networks. Instead, we want to simulate only a small constant number of hops per target link.

An extreme alternative would be to simulate only the target switch queue. This is inaccurate for two reasons. First, we need to preserve end-to-end round trip delays, as these affect the speed of the congestion control adaptation to congestion or its absence; hosts closer to the target adapt faster than those farther away. Second, we need to preserve the spacing of packets induced by the original topology—a large flow does not immediately dump all of its data into the queue for the target link; instead, those packets arrive spaced apart by the edge link capacity. Ignoring this effect would lead to larger queues and more delay at the simulated link than would occur at that link in the original network.

Thus, we construct a topology for each link-level simulation that reflects a performance-accuracy tradeoff, attempting to capture the most important effects for computing the delay contributed by the target link. Fig. 3.4 shows how topologies are minimized. The generated topology takes one of three shapes, depending on the location and direction of the target link: (i) a first-hop up-link from a host to a

ToR, (ii) a switch-to-switch link in the middle of the network, or (iii) a last-hop downlink from a ToR to a host.

Suppose the traffic through the target link originates from sources S and terminates in destinations T. In case `A` of Fig. 3.4, we connect the target link directly to each host in T via a dedicated link. If the target link is a switch-to-switch link (case `B`), we remove intermediate hops and connect the hosts in S directly to the input, and the output directly to the hosts in T. Lastly, if the target link is a last hop (case `C`), then the hosts in S are connected directly to the input. Rewriting the topology in this manner ensures that packets can traverse at most three hops, regardless of the size of the original topology.

MODELING ROUND-TRIP DELAY. Next, we set the link delays in each constructed topology to match the round trip delays in the original network. For example, in case `A` of Fig. 3.4, the round-trip time between host `0` and host `2` is 8 in both the original topology and the generated topology, even though `Parsimon` has removed intermediate hops between the switch and host `2`. Fig. 3.4 is meant as illustrative; as with ns-3, `Parsimon` can model arbitrary round-trip delays.

In data center networks, congestion controllers play a large role in determining the extent to which longer flows yield throughput to benefit the latency of short flows. Most algorithms such as DCTCP [6], DCQCN [97], and TIMELY [64] are *end-to-end* in the sense that sources adjust their sending rates based on feedback echoed from destinations [41]. With an end-to-end control loop, a source must wait an entire round-trip time (RTT) before being able to adapt its sending rate based on congestion feedback, resulting in longer queue lengths with higher RTTs. Thus, correctly modeling RTTs is essential to correctly modeling queue dynamics.

SELECTING LINK BANDWIDTHS. In some cases, we artificially increase the bandwidth of downstream links to ensure that they do not artificially add congestion. We say such links are *inflated*. For example, in cases `A` and `B` of Fig. 3.4, the bandwidths of the last-hop links are inflated. We want any queueing to be due to the target link and not the downstream link. By inflating downstream links, we remove store and forward delay (a small packet following a large packet would otherwise need to queue for the downstream link); it also addresses the case where core links are fatter than downstream links. Queueing at the downstream link itself is accounted for in case `C`. By contrast, we do *not* inflate first-hop links in cases `B` and `C`, as this would enable a long flow to arrive at the target link at a higher rate than it would in practice.

A cluster of sources sending simultaneously through an oversubscribed top-of-rack (ToR) switch in the original network will be throttled beyond what is implied by the edge link capacity. To improve

simulation speed, we ignore this effect and are therefore slightly conservative in our estimates for oversubscribed networks.

CORRECTING FOR ACK TRAFFIC. Since `Parsimon` only simulates one direction at a time, we must account for the load induced by acknowledgments due to traffic in the reverse direction. This is usually small, but can be significant at high load and where average packet size is small. Instead of modeling ACK traffic in detail, we apply a simple rule, mechanically reducing the forward bandwidth on each simulated link by the average volume consumed by ACKs for flows in the opposite direction over the course of the simulation. This correction is applied to all links but is most necessary for the target link. Note that `Parsimon` does not account for extra delay caused by ACK jitter on the reverse path; this could be an issue when applying our ideas to networks with bandwidth asymmetry between forward and reverse paths [13].

3.3.3 *Post-processing link-level results*

Each link-level simulation produces an FCT for each flow in the link-level workload, and these FCTs are used to compute delays. Recall from §3.3.2 that the delay is just the observed FCT minus the ideal FCT on an unloaded network. For each flow, we could, theoretically, estimate the end-to-end delay as some function of the delay contributed by each link for that flow. We discuss how that function works in `Parsimon`, along with its sources of bias, later in this section.

First, we address a different issue. Recall that we cluster similar links together (§3.4.2) so that we only simulate the flows through a single representative link for each cluster of links. Thus, to compute the end-to-end delay for a particular flow, we take a sample from the delay distributions at each hop in the path, or from the hop's standin representative.

In post-processing the link-level results and constructing these distributions, our primary objective is to support accurate estimates for *all flow sizes.* It is not enough to produce the correct FCT distribution across the entire workload; we must also accurately estimate the FCT distribution for short flows containing just a few packets as well as for long flows that last for hundreds of round trips. This extra requirement necessitates some post-processing before distributions can be constructed. Here we describe how this is done.

PACKET-NORMALIZED DELAY. Maintaining accuracy across all flow sizes would not be possible if we used delays directly. For example, long flows, which may experience variations in their bandwidth share over time, will almost always experience more absolute delay than short flows.

As a start, we can address this by normalizing delays by flow size: after computing the delay for a particular flow, we can then divide the delay by the flow's size in packets. We call the resulting metric the *packet-normalized delay*, and it has the intuitive interpretation of summarizing the flow's average delay per packet. Link-level distributions are constructed from packet-normalized delays rather than absolute delays. We normalize by the number of packets instead of the number of bytes because flows are discretized into—and therefore delays are incurred by—packets. Further, normalizing by the number of bytes loses accuracy for small flows, especially those smaller than the maximum packet size. For example, a 10 byte packet would be delayed by the same amount as would a 100 byte packet if it arrived in the switch queue just behind a jumbo (9 KB) frame [88].

BUCKETING DISTRIBUTIONS. Even with packet-normalized delays, we should still expect long flows to have different delay distributions than short flows. The FCT of a long flow is mainly determined by the throughput it achieves, while the FCT of a short flow depends on how much queueing it encounters. Further, congestion control algorithms trade the throughput of long flows for the latency of shorter ones to varying degree. An aggressive congestion control algorithm could try to keep queues near-empty [58], resulting in smaller short-flow delay and larger long-flow delay.

To ensure that estimates for different flow sizes are accurate, it is necessary to sample each packet-normalized delay from the appropriate distribution. We bucket the distribution of packet-normalized delays by flow size. Buckets need to contain enough samples to form statistically meaningful distributions, but they should also be small enough so that the values come from flows with similar delay characteristics (i. e.similarly-sized flows).

`Parsimon` uses a simple bucketing algorithm. In brief, we start with a packet-normalized delay per flow, and we sort them according to flow size. Then, starting with the shortest flow, we begin populating buckets. For each bucket b, let maxf_b and minf_b be the maximum and minimum flow sizes associated with b, respectively, and let n_b be the number of elements in b. Each bucket b apart from the last one is locally subject to two constraints

$$n_b \geq B \qquad \text{and} \qquad \text{maxf}_b \geq x * \text{minf}_b,$$

for some choice of B and x. Globally, `Parsimon` also ensures buckets are contiguous and non-overlapping. For any bucket, once the two local constraints are satisfied, `Parsimon` begins populating the next bucket, and the final bucket is assigned whatever elements remain.

In practice, we find $B = 100$ and $x = 2$ works well. Data center workloads have heavy-tailed flow size distributions in which short flows arrive much more frequently than long ones. With these parameters,

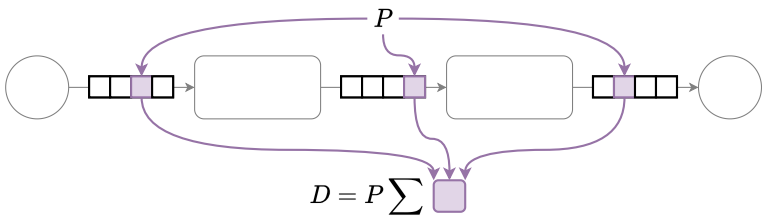


Figure 3.5: An illustration of how `Parsimon` aggregates link-level results into a path-level point estimate. `Parsimon` samples a packet-normalized delay (§3.3.3) from each link along the path, and combines these to estimate the end-to-end absolute delay D.

the first buckets will have size boundaries that are approximately powers of two, and as flows get larger, buckets will cover larger and larger ranges. This is the desired behavior. Intuitively, a queueing-sensitive 1 KB flow should not be grouped with a throughput-sensitive 1 GB flow, but a 1 GB flow can be grouped with a 10 GB flow provided the distribution of throughput is stable on long timescales. Accuracy across different flow sizes at finer or coarser resolution can be achieved by modulating x. We examined sensitivity to the number of buckets by decreasing x for selected experiments and found no meaningful change in the predicted distributions.

In summary, each link-level simulation produces FCTs, and these FCTs are used to construct bucketed distributions of packet-normalized delay. Since different links have different workloads, bucketing is performed on a per-link basis. This means that the links in any given path are likely to have different bucket sizes with different flow size ranges. In the next subsection (§3.3.4) we describe how the data are aggregated.

3.3.4 *Aggregating link-level estimates*

For any given range of flow sizes, the final distribution of (packet-normalized) delay for any path through the network can be estimated by selecting an appropriate distribution from each component link and then performing an n-ary convolution. However, the efficiency of this step must be considered. Since there are multiple distributions per link and potentially many paths through the network, performing all convolutions up front and storing one path-level distribution per path, per flow-size range would be costly in space and in time.

To avoid these costs, `Parsimon` uses an on-demand sampling strategy to perform the convolution. Recall that the simulation step resulted in bucketed distributions of packet-normalized delay per link, organized in a graph isomorphic to the original topology. `Parsimon` makes this graph a queryable object that is capable of supporting point estimates. Given a size, a source, and a destination, `Parsimon` computes a path

from the source to the destination and uses the size to select a distribution per-link. Then, one packet-normalized delay is sampled from each distribution and the results are subsequently combined into a point estimate. Suppose there are n hops and let $D_1^*, D_2^*, \ldots, D_n^*$ be the sampled packet-normalized delays. Then, the end-to-end absolute delay D is computed as

$$P \sum_{i=1}^{n} D_i^* \;=\; \sum_{i=1}^{n} D_i^* P \;=\; \sum_{i=1}^{n} D_i \;=\; D,$$

where P is the input flow size in packets and D_i is the absolute delay for hop i. Fig. 3.5 illustrates this process. Finally, to obtain a distribution of end-to-end delay estimates, we need only sample enough point estimates for the desired flow size range and source destination pairs.

3.3.5 *Primary source of speedup*

`Parsimon` speeds up large network simulations by considering the effect of each link in isolation, allowing it to scale in the size of the simulated network and the number of processing cores. Although the link is the unit of decomposition, `Parsimon`'s scaling ability is determined not by the total the number of links, but rather by the *fraction of total packets traversing any link.* In other words, `Parsimon`'s speed-up depends on the number of busy links and how well the load is balanced among them. This explains why `Parsimon` is most suited for large data center networks, where the total workload comprises many source destination pairs with many paths between them. If a network traffic is heavily skewed such that most of the workload traverses only a few paths, the amount of speedup will be limited.

3.3.6 *Primary sources of error*

To balance accuracy and performance, `Parsimon` makes a number of approximations, with some having more of an effect on accuracy than others. Here we catalog some of the main sources of error, describing 1) how we expect the errors to manifest and 2) what modifications, if any, could be made to address them.

BOTTLENECK FAN-IN. To simulate a given target link in the network, `Parsimon` constructs a topology that connects all of the source nodes feeding traffic directly into that target. In practice, of course, there would be multiple stages of fan-in, and that fan-in would tend to spread out any burst of arriving flows due to upstream bandwidth capacity constraints. Any target link would experience slightly less queueing and less congestion in reality than in `Parsimon`. Of course, `Parsimon` also simulates the upstream link; because it is closer to the

sources, its traffic and queueing behavior would be a closer model to what would happen in a full network-wide simulation.

Because `Parsimon` sums the delay contributed by each hop along a flow's path, the lack of fan-in will tend to slightly overestimate the delays caused by downstream links. Put another way, any delay induced by fan-in constraints is counted twice—once when we simulate the upstream link and again when we simulate the downstream link. In our evaluation, accuracy is slightly lower for networks with higher degrees of oversubscription, as we would expect. We could potentially remove this inaccuracy by including the upstream fan-in as part of the topology for each link simulation. Since simulation time is proportional to the number of hops, this would decrease individual link simulation efficiency by a small but significant factor.

LACK OF TRAFFIC SMOOTHING. Similarly, any cross-traffic that shares a portion of a path with traffic destined for the target link will tend to smooth out traffic before it reaches the target. `Parsimon` does not include any cross-traffic in its per-link simulation, making it slightly overestimate the queueing delay at the target link. Assuming the simulation is stable—that the arrival rate does not exceed the service rate for any link—the target link will experience the correct long-term average rate, but without as much smoothing as would happen in practice. We see evidence of this effect in our evaluation, where error is slightly larger for workloads with a predominance of short flows which would benefit more from smoothing. Of course, correctly modeling the effect of cross-traffic on the traffic arriving at a downstream link would be difficult to accomplish without reverting to a full network simulation.

LINK-LEVEL INDEPENDENCE. A more fundamental approximation is that link-level simulations are treated independently. This technique enables wholesale parallelization, but its accuracy depends on the amount of correlation between the traffic intensities on the various hops along the path. The more correlated the traffic, the more error `Parsimon`'s method produces.

Since `Parsimon` produces estimates by convolving delay distributions (adding independent random variables), full accuracy requires the mutual independence of delays among the links in every path. Consider a single-packet flow that traverses two hops, both with load l. If the delays along the two hops are independent, the probability that the flow will encounter *no queueing* is simply $(1-l)^2$. However, if both hops tend to have queueing at the same time (i. e.if the traffic intensities and therefore the delays are correlated), then that probability is closer to $1-l$. Since `Parsimon` does not distinguish between these two scenarios, the difference is not reflected in its estimates.

In very large networks with thousands of hosts and paths, and with realistic workloads, we expect the effects of correlation to be small. A basic result of queueing theory is that under some circumstances it is possible to analyze queues independently, even when the output of one queue connects to the input of another, so that queue behaviors are obviously correlated. One view of our work is that we are empirically observing that data center networks approximately admit product-form solutions for their equilibrium state queue distributions under realistic workloads.

However, some networks use PFC [97] to reduce packet loss due to go-back N error handling in some RDMA network interface cards. Because PFC suffers from head-of-line blocking, PFC can cause correlated congestion across multiple links, and so `Parsimon` would not be a good choice for modeling such networks. If correlation is a problem, we could potentially measure the degree of correlation and apply a correcting factor during the convolution step, but we leave that for future work.

ONE BOTTLENECK AT A TIME. Estimating the performance of long flows comes with an additional difficulty which is also exacerbated by correlated delays. While a single packet flow can only reside in one queue at a time, a long flow can be backlogged on multiple links *at the same time.* Depending on the specific congestion control mechanism, the throttling back of a long flow (the delay it experiences) is typically *not* the sum of the delays it would experience on individual links (as `Parsimon` approximates), but rather only the delay caused by the true (instantaneous) bottleneck. Since `Parsimon` sums all delays, it will overestimate the end-to-end delay for the long flow that encounters simultaneous cross-traffic congestion at multiple points along its path. In summary, `Parsimon` is more accurate when the congestion is episodic and temporary, appearing at different links at different times, and less accurate when congestion is persistent across multiple edge and core links of a given path.

Congestion on any link (and therefore simultaneous congestion on multiple links) becomes more common with higher network load, and we see this effect in our evaluation. We can potentially correct for this bias by using a more complex function for combining link delays when overall network utilization is high. Because network operators are often willing to over-provision their network hardware to reduce application tail latency, this is rare in practice. For example, some recent end-to-end congestion protocols, such as Homa [66], simply assume that network congestion predominantly occurs at the last hop of each path. We do not make such an assumption; we handle congestion equally wherever it might occur. However, we do assume that congestion events are not persistent and network wide.

Our approximations are biased toward producing overestimates rather than underestimates, because we expect network operators to be more sensitive to over-promising tail behavior, even if that comes at the cost of being too conservative with respect to capacity planning. Additional analyses on the errors induced by these approximations can be found in the appendix (§3.5.7).

3.4 COMPLEMENTARY METHODS

The previous section described how we decompose a single large network simulation into many small, independent ones that can be executed in parallel and later combined. This section describes additional optimizations that reduce, cluster, and prune these link-level simulations for better computational efficiency. These reduce the number of cores needed to simulate a given network within some time bound, or equivalently, the execution time on a single server machine.

3.4.1 *Fast link-level simulation*

By far the largest computational cost in `Parsimon` are the link-level simulations. Initially we used ns-3 as our link-level backend. However, as a general-purpose simulator, ns-3 is designed to support arbitrary protocols with arbitrary extensions, all the way down to hardware models. This is more flexible but means that every packet in ns-3 generates events at every host, queue, and link—as well as throughout the hosts' modeled network stacks.

Instead, we implemented a custom and minimal simulator optimized for high fidelity single link simulation. This backend only models the workload, topology, queueing, and congestion control. For congestion control, our prototype implements DCTCP's core algorithm [6] in a few tens of lines of code. For example, we do not need to model the mechanism for carrying ECN bits from switches back to endpoints. Switching to a custom simulator speeds up the individual link simulations by roughly an order of magnitude, with negligible loss of accuracy. Reducing the simulation time of the worst case (most congested) link also reduces the critical path dramatically. If more simulation features are needed, `Parsimon` can use ns-3 at the cost of using more cores.

3.4.2 *Clustering and pruning simulations*

Lastly, we recall that `Parsimon`'s decomposition results in two simulations per link: one in each direction (§3.3.1). On a large-scale 6,144-host topology we use for evaluation, there are over 9,000 links, and therefore over 18,000 simulations generated. Fortunately, data center topologies commonly induce symmetries that render some of these simulations

Algorithm 1 Greedy link clustering

```
unclustered ← AllLinks                          ▷ links here are unidirectional
clusters ← []                                   ▷ list of list of links
while not Empty(unclustered) do
    members ← []                                ▷ new cluster
    representative ← PopFirst(unclustered)
    Push(members, representative)               ▷ with initial member
    for candidate in unclustered do             ▷ find other members
        rfeature ← Feature(representative)
        cfeature ← Feature(candidate)
        if IsCloseEnough(rfeature, cfeature) then
            Push(members, candidate)            ▷ new member
            Remove(unclustered, candidate)
    Push(clusters, members)
return clusters
```

redundant. For example, up-links in the same ECMP grouping can be assumed to have the same characteristics and traffic patterns. Furthermore, the workloads themselves may also induce symmetries due to communication patterns and load balancing [74].

We can take advantage of these symmetries by clustering links that carry similar traffic and only simulating one representative from each cluster. Then, in each cluster, all links inherit the delay distribution produced by the representative link. `Parsimon`'s clustering requirement is quite specific, which limits the range of popular clustering algorithms that can be used. Let $l_1, l_2 \in L$ be any two link-level simulations, and let $d : L \times L \to \mathbb{R}$ be a distance function. Ideally,

$$l_1 \text{ and } l_2 \text{ are clustered together } \iff d(l_1, l_2) < \epsilon,$$

where ϵ is some bound. The left-to-right direction preserves accuracy; the right-to-left supports efficiency. Most centroid-based and density-based clustering algorithms aren't designed to provide the left-to-right property. Instead, `Parsimon` uses Alg. 1. This algorithm greedily clusters simulations together, using a distance function that predicts which links will have similar delay profiles. In our prototype, we check that the link flow size and inter-arrival time distributions—as well as their load levels—are close. We find this provides a reasonable tradeoff between efficiency and accuracy, but users can turn off the optimization at the cost of using more cores.

DISTANCE FUNCTIONS. To compute a distance between link loads, we compute the error. If a and b are two link loads, error e is computed as

$$e = \frac{|a - b|}{a}.$$

To compare distributions, there are many options. We opt for a function that is 1) easily interpretable, 2) scale-independent, and 3) ade-

quately captures differences in the tail. To compute a distance between two distributions, we extract 1,000 percentiles from each of them, and we compute a weighted mean absolute percentage error (WMAPE) between them. Suppose A and B are the sequences of extracted percentiles. Then, WMAPE is computed as

$$\text{WMAPE} = \frac{\sum_{i=1}^{n} |A_i - B_i|}{\sum_{i=1}^{n} |A_i|}.$$

For our purpose, A_i and B_i are non-negative for all i. We note it is a bit counterintuitive for our distance functions not to commute. However, we have found that it is easy to set thresholds for these metrics, and they produce adequate clustering for the workloads under study.

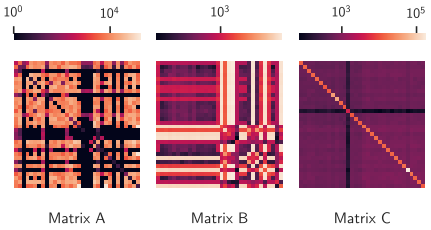
DISTANCE THRESHOLDS. Recall that we only want to cluster two links together if we expect their simulation outputs to be similar. Consequently, when setting a threshold for link loads we must consider the network and the workload being assessed. At high load, small differences in link loads can yield large differences in the tails of FCT distributions; in these cases, we typically set tighter thresholds to preserve accuracy (as usual, this is subject to a speed-accuracy tradeoff). For highly-loaded networks, we commonly require $e < 0.001$ or $e < 0.002$ for links to be clustered together. Ideally, this decision would be made on a link-by-link basis, so that tighter thresholds would be set only for high-load links—doing so may allow for more liberal clustering of the low-load links contributing little delay, resulting in more pruned simulations. However, the current prototype sets a single threshold per simulation. To set a threshold between distributions, we typically require WMAPE < 0.1.

3.5 EVALUATION

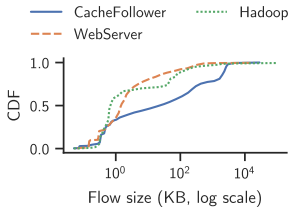
`Parsimon`'s goal is to quickly estimate tail latencies for a variety of large data center networks and workloads. In evaluating `Parsimon`, we would like to assess 1) `Parsimon`'s accuracy and performance at the scale of thousands of hosts, and 2) how accuracy is affected by a wide range of variables over the workload and the topology.

Our strategy is as follows. Using workloads extracted from industry datasets, we start with a 384-rack, 6144-host topology to evaluate `Parsimon`'s speed and accuracy in one scenario at scale. Then, to evaluate nearly 200 other topology and workload scenarios, we downsample the workload so that it can run on a smaller 256-host topology. This allows us to run enough ns-3 simulations quickly enough to perform a detailed sensitivity analysis.

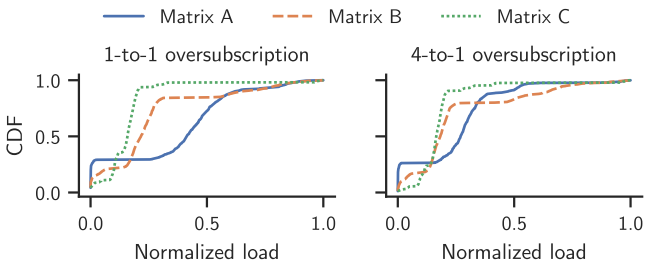
To more clearly illustrate sources of error in `Parsimon`, we also construct and evaluate `Parsimon` on synthetic workloads on a small-scale parking lot topology in Appendix §3.5.7.



(a) Traffic matrices (32-rack sample)



(b) Flow size distributions



(c) Normalized link load distributions

Figure 3.6: In the evaluation, we model workloads using data from Roy *et al.*'s study of Meta's data center network [74]. The traffic matrices in Fig. 3.6a are extracted from the accompanying dataset, and the flow size distributions in Fig. 3.6b are estimated from the published data. Lastly, for a given topology, the distribution of link loads depends on 1) the traffic matrix and 2) the degree of oversubscription. Fig. 3.6c shows the link loads induced by the matrices in Fig. 3.6a on two 32-rack topologies with different overprovisioning. The x-axis is normalized to the maximum link load.

3.5.1 *General setup*

Each scenario we consider has six components: 1) a topology size, 2) an oversubscription factor, 3) a traffic matrix, 4) a flow size distribution, 5) a burstiness level, and 6) a maximum load level. Here, we briefly describe how these are specified and configured. We also discuss which `Parsimon` variants we will assess and how we establish a baseline.

TOPOLOGY AND OVERSUBSCRIPTION. To mimic an industry topology, our topologies are modeled after Meta's data center fabric [9]. In brief, there are three layers of switches: hosts connected to a top-of-rack switch (ToR) with 10 Gbps links constitute a *rack*, racks connected to each other via fabric switches with 40 Gbps links constitute a *pod*, and pods connected to each other via spine switches with 40 Gbps links constitute a *cluster*. Spine switches are organized in *planes*. We can modulate the size of a topology (corresponding to a cluster) by adjusting the number of pods, the number of racks per pod, and the

Variant	Clustering?	Link-level backend
`Parsimon`	No	custom
`Parsimon/C`	Yes	custom
`Parsimon/ns-3`	No	ns-3
`Parsimon/inf`	—	custom

Table 3.1: The `Parsimon` variants under consideration. `Parsimon/inf` is a variant that assumes infinite cores and memory.

number of hosts per rack, and we can modulate the oversubscription factor by adjusting the number of spines per plane.

TRAFFIC MATRICES. The traffic matrices are extracted from the datasets accompanying Roy *et al.*'s study of Meta's data center network [74]. The data only allow us to construct reliable rack-to-rack matrices. When sampling workloads, we use the matrices to generate rack-to-rack traffic, but once a rack is chosen, we select its hosts uniformly at random. This may bear semblance to reality: according to Roy *et al.*, Meta's racks typically only contain servers in the same role, and load balancing is used pervasively. We use traffic matrices from three different clusters: a database cluster (matrix A), a web server cluster (matrix B), and a Hadoop cluster (matrix C). Fig. 3.6a shows 32-rack samples of the matrices.

FLOW SIZES AND BURSTINESS. We use three flow size distributions, estimated from published data in Roy *et al.*'s study [74]. These are reproduced in Fig. 3.6b. For inter-arrival times, we use the log-normal distribution to model bursty traffic, and we modulate the burstiness by adjusting the log-normal shape parameter σ. For low burstiness, we select $\sigma = 1$, and for high burstiness, we choose $\sigma = 2$.

MAXIMUM LOAD LEVEL. When setting a load level, we ensure that the offered rate is less than the link capacity for each link by specifying the maximum load level that any link can have. Note that a given maximum load level may result in different link load distributions, depending on the traffic matrix and the topology. Fig. 3.6c shows the distribution of normalized link loads on a 32-rack topology with the traffic matrices in Fig. 3.6a and two different oversubscription factors. When describing how loaded a topology is, we will usually specify the average load of the top 10% most loaded links.

`parsimon` VARIANTS AND BASELINE. To establish a baseline for `Parsimon`'s accuracy and performance, we use ns-3 with the optimized build profile. We also consider several `Parsimon` variants, summarized in Table 3.1. By default, `Parsimon` uses the custom link-level

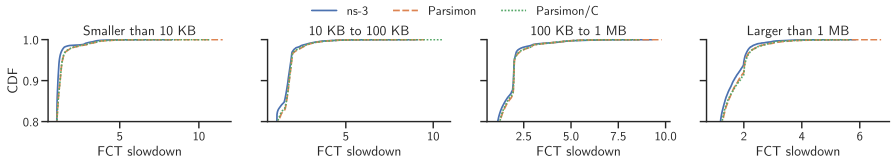


Figure 3.7: CDFs of FCT slowdown estimated by ns-3 and two `Parsimon` variants (note the y-axis). On a large network with 6,144 hosts, an industry traffic matrix (matrix B), and 2-to-1 oversubscription in the core, `Parsimon`'s latency estimates are similar to those produced by full-fidelity simulation. Table 3.2 shows the performance of each estimator.

Estimator	Time	Speed-up
ns-3	10h 48m 26s	—
`Parsimon`	4m 13s	154×
`Parsimon/C`	1m 19s	492×
`Parsimon/inf`	21s	1864×

Table 3.2: Running times and speed-up of `Parsimon` variants for five seconds of simulated time on a large oversubscribed network with thousands of hosts. We find that `Parsimon` estimates latencies orders of magnitude faster than does ns-3. If there is ample opportunity for clustering or if there are infinite compute resources, speed-up is substantially further increased. Measurements were taken on a 32-core machine.

backend (§3.4.1) with clustering turned off. This expresses a lower bound on `Parsimon`'s expected speed-up given a particular machine. `Parsimon/C` adds clustering to the default variant using the methods described at the end of §3.4.2, and `Parsimon/ns-3` replaces the default's custom backend with ns-3. Lastly, `Parsimon/inf` provides an estimate of `Parsimon`'s performance given infinite cores and infinite memory, computed by adding the run time of the longest link-level simulation to the fixed costs of network setup and convolution sampling. This represents an upper bound on the `Parsimon`'s achievable performance. All performance measurements are taken on a 32-core AMD Ryzen Threadripper 3970X.

3.5.2 *Analysis on a large-scale network*

Here we evaluate `Parsimon`'s accuracy and performance on a 384-rack, 6144-host topology. The topology has eight pods, 48 racks per pod, and 16 hosts per rack, with 2-to-1 oversubscription. For the workload, we use matrix B, the WebServer flow size distribution, and high burstiness ($\sigma = 2$). We set a maximum link load of about 50%, which gives the 100 most loaded links an average load of 32%, and the top

10% most loaded links an average load of about 15%. We configure all simulations to run for five seconds of simulated time. To establish a baseline, we first run the scenario in ns-3, then we run the scenario in Parsimon and Parsimon/C (see Table 3.1). Due to memory constraints we omit Parsimon/ns-3 here, but we include its analysis at smaller scale in §3.5.3.

Fig. 3.7 shows the accuracy of Parsimon relative to ns-3 across four flow size bins. We find that across all bins, both variants accurately estimate tail latencies. If we consider all flow sizes together, we find that Parsimon and Parsimon/C overestimate the p99 FCT slowdown by 8.8% and 7.5%, respectively.

Table 3.2 shows the running time and speed-up for each estimator, which includes topology generation and convolution sampling overheads where applicable. While ns-3 took nearly 11 hours, Parsimon without clustering took four minutes and 13 seconds, for a speed-up of 154×. If we turn clustering on by using Parsimon/C, the running time is further reduced to one minute and 19 seconds, for a speed-up of 492×. [1] In this case, only 25% of links were simulated; the rest were pruned. Lastly, Parsimon/inf estimates Parsimon's best possible performance given infinite compute resources. The longest-running single-link simulation took 11 seconds, and with the additional 10 seconds required for network setup and convolution sampling, the fastest projected running time is 21 seconds.

We chose an oversubscribed topology to slightly disadvantage Parsimon's method, as oversubscription can lower Parsimon's accuracy. §3.5.3 analyzes the effect of oversubscription in more detail. We also ran the above experiment on a topology without oversubscription, which for the same maximum load setting increased the top 10% average link load from 15% to 25%. We found Parsimon's p99 accuracy improved from 9% to about 7%, while Parsimon/C's accuracy remained approximately the same. However, because aggregate load increased, ns-3 took 27 hours for five seconds of simulated time, and speed-ups for Parsimon, Parsimon/C and Parsimon/inf were 152×, 872×, and 3487×, respectively. Parsimon/C benefited from the increased number of links in each ECMP grouping, allowing it to prune 85% of the link-level simulations.

3.5.3 *Sensitivity analysis at small scale*

Next we turn our attention to how different aspects of workloads and topologies affect Parsimon's accuracy. To be able to simulate enough scenarios in ns-3 for a sensitivity analysis, we downsample the topolo-

1 We advise caution both in interpreting this number and in generalizing it to scenarios at large. While our workloads are modeled after industry data, they are still synthetic. There may be more or less opportunity to cluster and prune link-level simulations, depending on the structure of real workloads and the quality of the clustering algorithm.

Parameter	Sample space
Oversubscription	1-to-1, 2-to-1, 4-to-1
Traffic matrix	Matrix A, Matrix B, Matrix C
Flow size distribution	CacheFollower, WebServer, Hadoop
Burstiness	Low ($\sigma = 1$), High ($\sigma = 2$)
Max load	26% to 83% (continuous range)

Table 3.3: The sample space for the sensitivity analysis in §3.5.3.

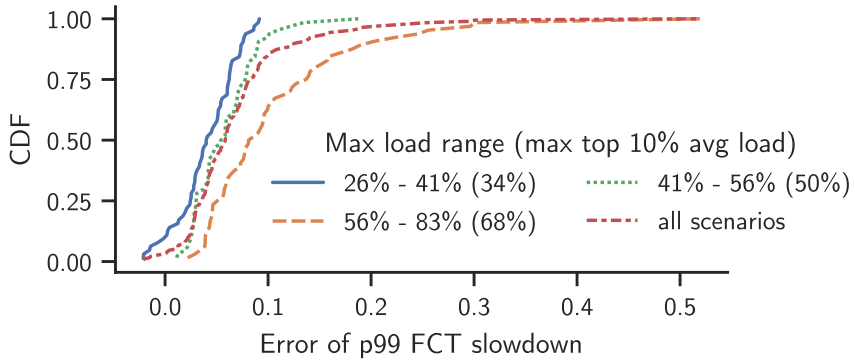
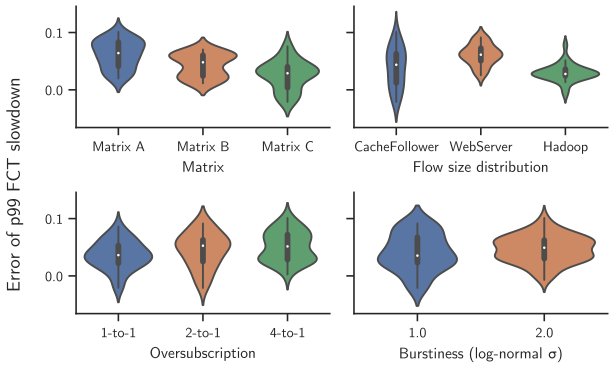


Figure 3.8: CDFs of p99 error between `Parsimon` and ns-3 across all scenarios drawn from the sample space in Table 3.3. The distributions are binned by maximum load. In parentheses, we give the maximum value for the top 10% average load in each bin. Under common conditions of low to moderate load, `Parsimon`'s estimates for the p99 FCT slowdown are reliably within 10% of the ground truth.

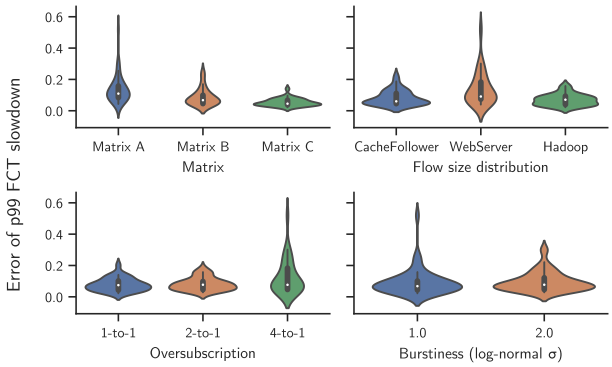
gies and traffic matrices to 32 racks. The resulting topologies have two pods, 16 racks per pod, and eight hosts per rack, and the number of spines per plane varies to accommodate different oversubscription factors.

Our approach is as follows. First, we construct a sample space over the parameters defining the workload and the topology (aside from the number of servers, which is fixed). The sample space is shown in Table 3.3. Then, we sample 192 scenarios uniformly at random, and we run ns-3 and the default `Parsimon` variant on each of them for several seconds of simulated time. Next, for each scenario, we take the p99 FCT slowdown estimated by both ns-3 and `Parsimon`, and we compute the error between them. If these values are n and p respectively, then the error is $(p - n)/n$. Negative values indicate that `Parsimon` produced an underestimate.

Since we have one error value per scenario, the errors give rise to distributions of error associated with the original sample space. Now what remains is to determine how the workload and topology parameters affect error distributions. To start, recall from the discussion in §3.3.6 that the magnitude of error is expected to be load-dependent,



(a) Max load $\leq$ 50%



(b) Max load $>$ 50%

Figure 3.9: Distributions of p99 error between `Parsimon` and ns-3, faceted by different workload and topology parameters. For each distribution we show the median, the quartiles, and a rotated kernel density estimation. We consider the low-load regime (Fig. 3.9a) and the high-load regime (Fig. 3.9b) separately. At low load, the workload and topology parameters only have a modest effect on `Parsimon`'s accuracy, but at high load, the conditions leading to the largest errors come into view: high load, high oversubscription, with very short flows. Note the different y-axes between the two load regimes.

with higher errors typically manifesting at higher loads, so we begin by examining the effect of the maximum load setting on `Parsimon`'s accuracy.

MAXIMUM LOAD. Fig. 3.8 shows the error distributions binned by maximum load. Among all scenarios, `Parsimon`'s p99 estimates are within 10% of ns-3's estimates 85% of the time At high load, we observe larger overestimates of up to 52% in the worst case. In the most highly-loaded group of scenarios—with maximum link loads between 56% and 83%—`Parsimon` is within 10% of ns-3 62% of the time, with an average error of about 11%. However, this includes scenarios where 10% of the links have an average load of up to 68%, which is much higher than what is reported in the literature. For

Error	Max load	Matrix	Sizes	Oversub	σ
51.9%	77.6%	A	WebServer	4-to-1	1
30.1%	67.3%	A	WebServer	4-to-1	2
29.6%	67.0%	A	WebServer	4-to-1	2
25.6%	65.9%	A	WebServer	4-to-1	1
24.6%	73.2%	B	WebServer	4-to-1	1

Table 3.4: The five scenarios with the highest error values from the sensitivity analysis in §3.5.3.

example, Roy *et al.* report that in Meta's data center network, 99% of host links are less than 10% loaded, and the top 5% of core links have loads between 23% and 46% [74]. Among scenarios where the maximum link load is between 26% and 41%, `Parsimon` is within 10% of ns-3 100% of the time. If we further include scenarios with maximum link loads between 41% and 56%, that number falls to 96%. Finally, while `Parsimon`'s techniques tend to overestimate latencies, in 3% of the scenarios, `Parsimon` underestimates p99 slowdown by up to 2%.

OTHER PARAMETERS. We next turn to the effects of all other workload and topology parameters. We start by only considering scenarios where the maximum link load is less than or equal to 50%; this will tell us whether any of the parameters have a large effect on accuracy in the low-load regime. Fig. 3.9a shows the median error and error distributions as a violin plot for low-load scenarios grouped by traffic matrix, flow size distribution, oversubscription, and burstiness. Overall, changes to these parameters appear only to have a modest effect. The choice of traffic matrix has the clearest trend, but load is a confounder here: recall from Fig. 3.6c that different traffic matrices yield different link load distributions for the same maximum load setting.

When we look at the high load regime in Fig. 3.9b, a clear picture comes into view. We see much longer tails in error distributions for matrix A, the WebServer flow size distribution, and 4-to-1 oversubscription. Together with Fig. 3.9a, this suggests that none of these settings has a strong effect on its own, but *coupled together in the high load regime,* they have a pronounced effect on `Parsimon`'s accuracy. Matrix A induces higher average load and has more cross-rack traffic, making it more likely for its flows to encounter multiple simultaneous bottlenecks. The WebServer flow size distribution is dominated by short flows (Fig. 3.6b), a third of which are smaller than 1 KB and 80% of which are smaller than 10 KB. Because more of the traffic completes within a single round trip, there is more ephemeral congestion and bandwidth smoothing can have a larger impact.

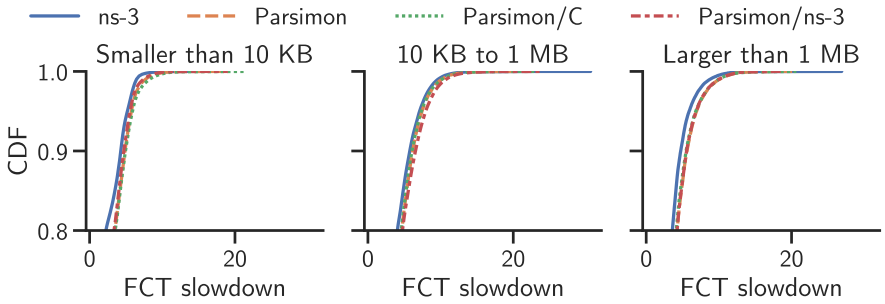


Figure 3.10: CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the scenario whose error is at the 85th percentile of the p99 error distribution. Note the y-axis. Even though the accuracy here is worse than in the common case, `Parsimon`'s estimates remain close across most of the tail. Also shown is `Parsimon/ns-3`.

Finally, oversubscription has an effect at high load: if we removed all scenarios with 4-to-1 oversubscription, the maximum error would only be 20% rather than 52%, even at high load. In addition to the double counting of delays described in §3.3.6, oversubscription can also increase correlations in link delays. To achieve 4-to-1 oversubscription in topologies as small as these, there are only four spine switches per plane forwarding traffic between groups of 16 racks, leaving relatively few paths through the core. Fewer paths can result in higher degrees of correlation—especially with matrix A, whose traffic is primarily inter-rack (Fig. 3.6a). Finally, this setting combined with the short flows from the WebServer distributions gives rise to errors of up to 52%.

Table 3.4 lists the scenarios with the top five highest error values. Four have matrix A, all have the WebServer distribution, and all five have 4-to-1 oversubscription. In this group, the average maximum load is 70.2%. Since we expect the combination of all-to-all workload, heavily oversubscribed topology, and persistently high core utilization to occur relatively infrequently, the data suggest that `Parsimon` maintains good accuracy under common conditions.

3.5.4 *Analysis of one configuration*

We pick one representative scenario to examine in more detail, to test if our approach is robust to alternate definitions of tail latency, congestion control protocol, workload, and topology. To pick a scenario whose accuracy is somewhat worse than the average case, we rank-order all scenarios by error and select the one at the 85th percentile. This has matrix A, the Hadoop flow size distribution, low burstiness, 2-to-1 oversubscription, and a maximum load of 68% (with a top 10% average load of 56%).

Protocol	Max load	Error in p99 FCT slowdown		
		< 10 KB	10 KB - 1 MB	> 1 MB
DCTCP	45%	1.4%	9.2%	15.9%
TIMELY	45%	4.0%	17.9%	13.7%
DCQCN	45%	5.9%	11.6%	12.8%
DCTCP	56%	2.8%	9.2%	14.6%
TIMELY	56%	8.1%	20.0%	11.3%
DCQCN	56%	7.6%	14.6%	12.2%
DCTCP	67%	13.8%	11.3%	13.6%
TIMELY	67%	13.3%	18.2%	5.0%
DCQCN	67%	18.0%	15.2%	13.6%

Table 3.5: Prediction error of `Parsimon`/ns-3 for estimated p99 FCT slowdown with three different congestion control protocols for the sample configuration at different load levels and for different request sizes.

TAIL DISTRIBUTION. Operators may differ in their definitions of tail latency, e.g., focusing on the 90th or 99.9th percentile, rather than just the 99th FCT slowdown. Fig. 3.10 shows the tail of the cumulative distribution of FCT slowdown for different flow sizes for the selected configuration, for ns-3 and each of the `Parsimon` variants. The prediction error is similar across the tail of the distribution for this scenario, with little accuracy difference between any of the variants.

TRANSPORT PROTOCOLS. We use the sample scenario to test the generality of `Parsimon` to two additional congestion control protocols, DCQCN [97] and TIMELY [64]. DCQCN is designed for RDMA traffic, while TIMELY uses network delay, rather than ECN signals, to detect congestion. To focus on prediction error for our approximation methods, we use the pre-existing ns-3 implementation of the protocols as the `Parsimon` link level simulator for this part of the evaluation. Note that `Parsimon` and `Parsimon`/ns-3 exhibit a few percent difference in p99 error for DCTCP for this configuration. Because the prediction error for different congestion control protocols may depend on the amount of congestion, we also run the experiment at varying load levels.

Table 3.5 shows the prediction error for `Parsimon`/ns-3 relative to ns-3 in the estimated p99 FCT slowdown at three load levels for the three transport protocols, aggregated by request size. For this configuration, `Parsimon` is most accurate for small flows and low to moderate maximum link utilization, and that is true for all three congestion control protocols. DCTCP has somewhat lower error for small and medium size flows at low to moderate utilization. Relative error is

Name	Matrix	Sizes	Max load	σ
W0	A	CacheFollower	~20%	2
W1	B	WebServer	~20%	2
W2	C	Hadoop	~20%	2

Table 3.6: The three workloads mixed together in §3.5.5.

higher for larger transfers and higher maximum link utilization, with no clear pattern in the error for different protocols.

3.5.5 *Mixed workloads*

`Parsimon`'s methods are designed to estimate performance distributions rather than per-flow metrics. However, it is often useful to aggregate FCT performance estimates in different ways. For example, an operator may wish to estimate the performance of individual virtual networks or individual services. In this section, we conduct a simple experiment to assess `Parsimon`'s ability to estimate performance for separate aggregates.

We start by mixing three different workloads—each with its own traffic matrix and flow size distribution—into one workload. Table 3.6 summarizes their differences. Each workload has a maximum load setting of 20% and a high burstiness setting ($\sigma = 2$), and their combination results in a maximum link load of about 50%. We run the combined workload on the small-scale topology with 2-to-1 oversubscription from §3.5.3, and we observe the accuracy for each workload faceted by flow size. Fig. 3.11 shows the cumulative distribution function (CDF) of FCT slowdown for ns-3 and `Parsimon`. We observe that across all workloads and flow size bins, `Parsimon` maintains good accuracy.

3.5.6 *Link failures*

One operational use case for `Parsimon` is to estimate counter-factual network performance in the presence of potential link failures or planned outages. In this section, we use the sample scenario from §3.5.4 (matrix A, the Hadoop flow size distribution, low burstiness, 2-to-1 oversubscription, and a maximum link load of 68%) to evaluate `Parsimon` for this use case. For this configuration, the error in estimated p99 FCT slowdown between ns-3 and `Parsimon` was around 10%. Since link failures increase the load on the remaining links, we should expect some decreased accuracy for `Parsimon` in this case. On the other hand, simulating all possible network failures in ns-3 would be prohibitively expensive.

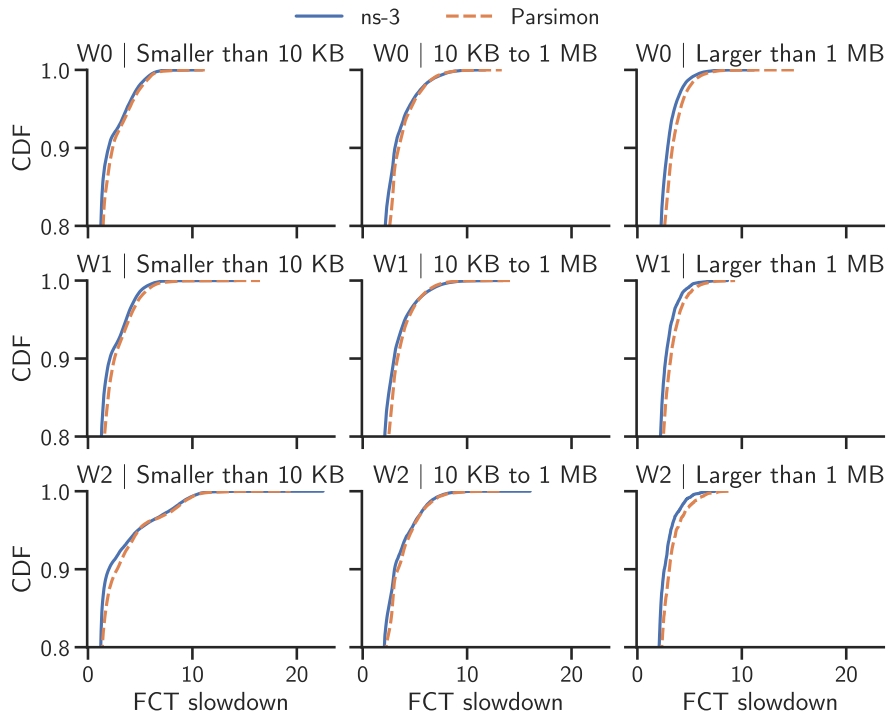
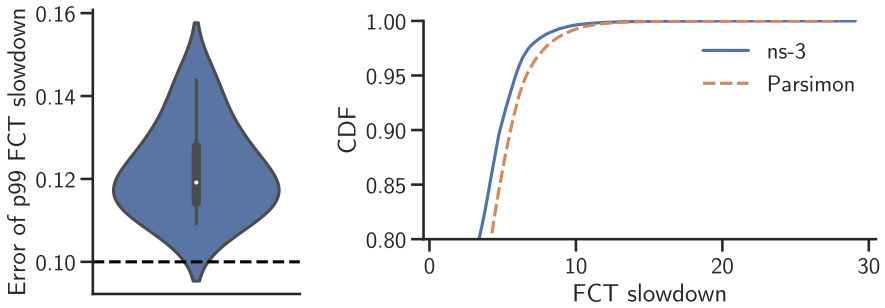


Figure 3.11: CDFs of FCT slowdown for ns-3 and `Parsimon`, bucketed by workload and flow size. Note the y-axes. When mixing workloads in a single simulation, `Parsimon` can accurately estimate performance distributions for individual workloads in addition to full-network aggregates.

In selecting links to fail, we only consider links in ECMP groupings, such that the failure of one link causes traffic to be routed to the other links in the group. In Meta's data center fabric [9], this corresponds to links between fabric switches and spine switches and links between ToR switches and fabric switches. In the small 32-rack topology used here (§3.5.3 for details), there are 96 such links. We run ten trials, each time picking a random one of the links to fail, keeping the workload constant. We note that this setting represents a particularly bad case for `Parsimon`: in addition to the high link loads, the scenario uses an all-to-all communication pattern on a small and oversubscribed topology, which means each link failure in the core can have an outsized effect on other core links.

Fig. 3.12a shows the distribution of errors in p99 estimates. With a single link failure, the errors range from 11% to 14%, with a median error of 12%. Fig. 3.12b shows the estimated CDFs of FCT slowdown for the trial with the highest error.



(a) p99 errors (b) CDF with the max p99 error (0.144)

Figure 3.12: Errors between ns-3 and `Parsimon` in estimated FCT slowdowns when there is a link failure. Fig. 3.12a shows the error distribution for p99 estimates from ten trials—each with one random link failure—with the dashed line showing the error with no link failure. Fig. 3.12b shows the CDF of FCT slowdowns for the trial with the highest p99 error. For the small oversubscribed topology used in this experiment, a link failure modestly increases estimation error.

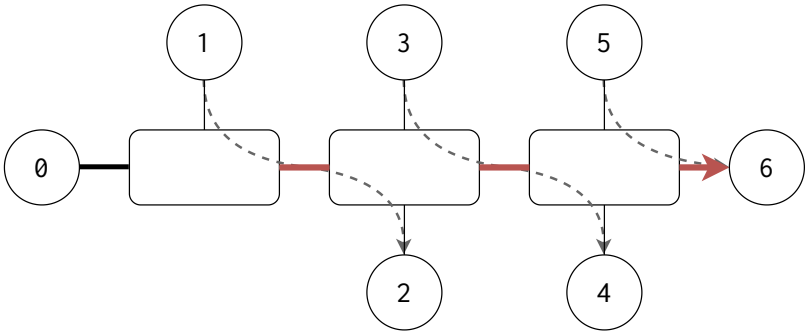


Figure 3.13: The parking lot topology used in §3.5.7. In this topology, zero sends to six, one sends to two, three sends to four, and five sends to six. We refer to the traffic from zero to six as *main traffic* and to all other traffic as *cross traffic*. The bolded red links contain both main traffic and cross traffic, and we call them *congested links*.

3.5.7 *Studying error sources*

Recall from §3.3.6 that `Parsimon`'s approximations induce errors in its end-to-end estimates. In this section, we use microbenchmarks to study the effects of some pathological cases on `Parsimon`'s accuracy. For an initial discussion on these topics, please refer to §3.3.6.

Throughout, we use the parking lot topology shown in Fig. 3.13 with 40 Gbps links. The flow of traffic through the topology is shown with arrows and described in the caption. We refer to the traffic from node zero to node six as *main traffic* and to all other traffic as *cross traffic*. The bolded red links contain both main traffic and cross traffic, and we call them *congested links*. In all experiments, we set the load of the main traffic to 25%. When there is cross traffic, its load is also

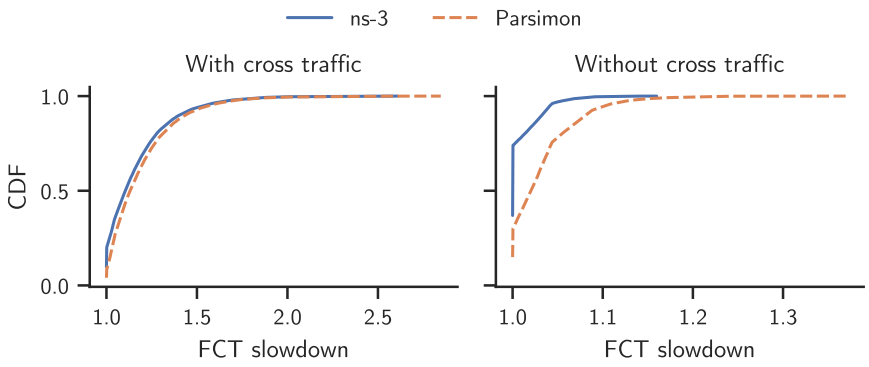


Figure 3.14: CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic, both with and without cross traffic. When there is cross traffic, errors arising from first-hop delays are second-order, as most delays are cause by queueing on the congested links. However, when there is no cross traffic, those errors become dominant. The graph on the right uses the same workload as the one on the left, except the cross traffic is removed. Note the different x-axes.

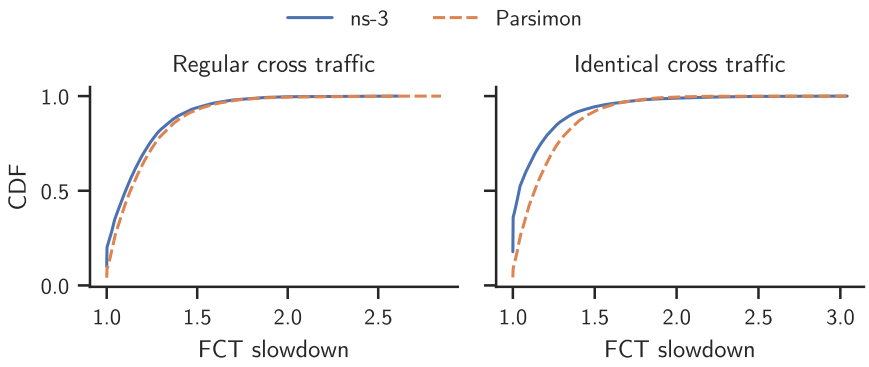
25%, yielding a total load of 50% on all three congested links. Lastly, to isolate the effects on the main path from zero to six, we measure FCT slowdown distributions only for the main traffic.

3.5.7.1 *First-hop delays*

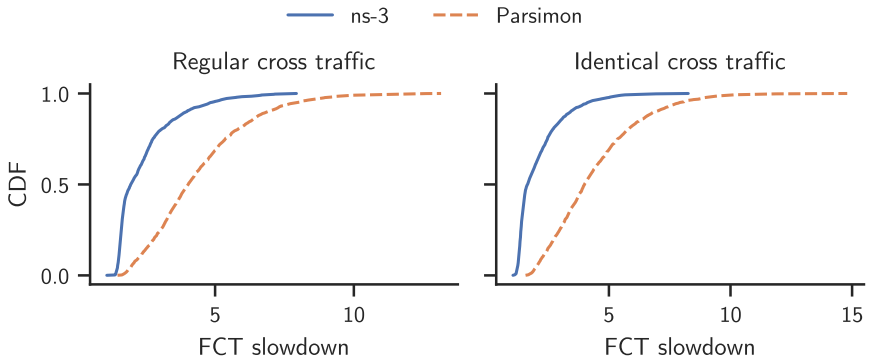
First, consider the case where all traffic in Fig. 3.13 originates from node zero and is destined to node six, and recall that all links have the same capacity. In a real network, all queueing in this scenario would occur at the first hop. Subsequent hops would see traffic completely smoothed, and they would therefore contributing zero queueing delay.

If we re-examine how link-level topologies are constructed in Fig. 3.4, we see that this smoothing effect is captured, since all traffic passes through edge links with the original edge-link capacities. However, for the link-level topologies in cases B and C of Fig. 3.4, it is possible for first-hop edge links to contribute delays that will be (erroniously) attributed to the target link. In most cases, we expect the magnitude of this error to be small. A target link will almost always have multiple sources, and only the traffic passing through the target link is simulated. Consequently, the first-hop delays in link-level simulation are expected to be small compared to delays accrued at target links.

The scenario which we first described—in which all traffic on a path originates from a single source—represents the worst case. Here, all target links (aside from the first hop) contribute no queueing delay, thus magnifying the error induced by repeatedly counting the first-hop delays for each target link. Fig. 3.14 shows this effect. In this experiment, the main traffic consists of one kilobyte flows, and the cross traffic consists of 10 kilobyte flows. All traffic follows a Poisson arrival process. With cross traffic, we see from the graph on the left



(a) Short flows (1 KB), Poisson cross traffic



(b) Long flows (400 KB), Poisson cross traffic

Figure 3.15: CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic with regular or identical cross traffic. The main traffic consists either of short flows (Fig. 3.15a) or long flows (Fig. 3.15b). When delays are artifically correlated by replicating the same cross traffic across hosts, accuracy decreases for both short and long flows, with long flows seeing larger errors. In fact, long-flow estimates have significant error even when delays are not explicitly correlated; this is due to the simultaneous delays induced by the smooth Poisson cross traffic.

that `Parsimon` accurately estimates the FCT slowdown distribution of the main traffic. However, when we remove the cross traffic, as done to produce the graph on the right, we see substantial error in `Parsimon`'s estimates due to the first-hop delays previously described. We note that this error exists *even when there is cross traffic*, but the error contributes so little to total delays—which are dominated by queueing at congested links—that `Parsimon` still maintains good accuracy.

3.5.7.2 *Correlated and simultaneous delays*

Next we examine the effect of correlated and simultaneous delays on `Parsimon`'s accuracy. We begin by artificially correlating delays and examining the effect on estimated slowdown distributions. Note that

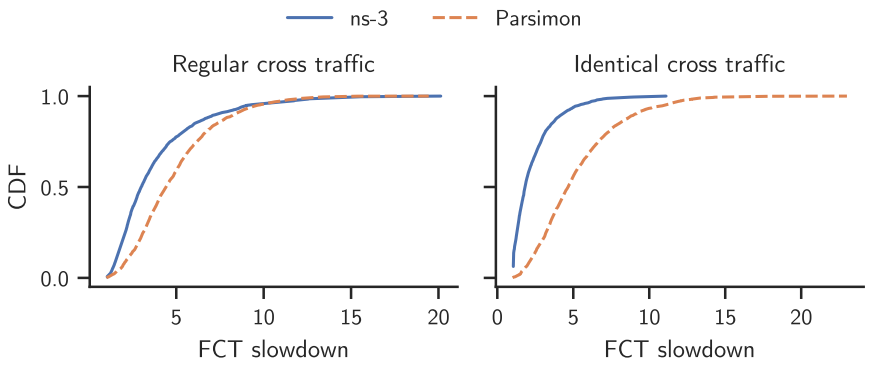


Figure 3.16: CDFs of FCT slowdown for the same scenario as in Fig. 3.15b, but with bursty cross traffic (log-normal inter-arrival times, $\sigma = 2$). When the cross traffic is bursty, long flows experience fewer simultaneous delays with regular cross traffic. This results in less error in `Parsimon`'s estimates.

if the delays along a path are positively correlated—for example, if the probability of encountering delay at hop $i+1$ is higher given there is delay at hop i—then we also expect to see more simultaneous delays along the path. We create these correlated delays by modulating the cross traffic. For regular unmodified cross traffic, we use the same setup as in the previous subsection (§3.5.7.1). To artificially correlate delays, we replicate the exact sequence of flows from source one on sources three and five in Fig. 3.13, so that all three sources of cross traffic send the same flows at the same time. This produces an extreme case of correlation.

Because short-flow and long-flow estimates have different sources of error, we separate the two cases when generating the main traffic. For short flows we use the same one kilobyte flows as before, and for long flows we generate flows that are 10 times the maximum bandwidth-delay product, or 400 kilobytes. Fig. 3.15 shows the effect of correlating delays on `Parsimon`'s accuracy for short and long flows.

SHORT-FLOW MAIN TRAFFIC. In the case of short flows (Fig. 3.15a), a chief effect of increased correlation is to alter the probability that a flow will encounter queueing. For example, suppose a short flow traverses only two links at 50% utilization. If the delays of the two links are independent, we can estimate the probability that the flow encounters no delay (i. e.no queueing) as $50\% \times 50\% = 25\%$. However, if the delays are perfectly positively correlated, then the probability that the flow encounters no delay increases to 50%. `Parsimon` does not capture this effect because it treats all links independently; in this experiment, this manifests as slight overestimates in FCT slowdown distributions.

LONG-FLOW MAIN TRAFFIC. While the total delay for a short flow can be thought of as the sum of individual link delays, the same

reasoning does not straightforwardly extend to long flows. Unlike a short flow, a long flow occupies multiple hops at the same time, and only the bottleneck at each instant contributes to end-to-end delay. Summing link delays is therefore only appropriate if different hops contribute significant delays at largely different times. However, `Parsimon` always aggregates individual link contributions by adding them, regardless of whether a link was the bottleneck when the delay was incurred. When we turn our attention to Fig. 3.15b, we see that not only is the effect of identical cross traffic more severe, but also there is significant error even with regular cross traffic. This is because the cross traffic is smooth (recall that it uses uniform flow sizes and a Poisson arrival process). Smooth traffic results in small but frequent delays at congested links, increasing the chance that long flows will experience simultaneous delays.

In Fig. 3.16, we duplicate the scenario in Fig. 3.15b, except we make the cross traffic bursty by using a log-normal inter-arrival time distribution with shape parameter $\sigma = 2$. Because the cross traffic is bursty, there is less simultaneous delay in the regular case, and the induced error is less dominant. Consequently, `Parsimon`'s estimates are closer to the ground truth in the graph on the left. Identical cross traffic still induces large and frequent simultaneous delays, so the errors remain in the graph on the right.

3.6 CONCLUSION

In this chapter, we propose and evaluate a new method for computing a conservative estimate of flow-level tail latency for large scale data center networks, given an arbitrary traffic matrix, topology, flow size distribution, and inter-arrival process. Our approach decomposes the problem into a large number of individual link simulations, specially constructed to produce accurate estimates of the probability distribution of delay contributed by congestion at each link. We then mechanically combine these link-level delay distributions to produce flow-level estimates. On a large-scale network using a commercial workload, our approach outperforms ns-3 by a factor of 492 on a single multicore server with a loss of accuracy of less than 9% in the tail of the latency distribution.

More importantly for this thesis, `Parsimon`'s predictions are fast enough to enable *closed-loop reconfiguration* rather than just offline what-if analysis. In subsequent collaboration with colleagues at MIT, we helped develop `m3` [56], which speeds up predictions and addresses some of `Parsimon`'s limitations by raising the abstraction from packets to flows, decomposing by paths rather than links to capture path-level effects, and using machine learning to bridge the gap between simulation and real system behavior. The result is a predictor that is both faster and more accurate than `Parsimon` on real deployments.

In the next chapter, we embed these predictors inside `Polyphony`, a controller that fuses model predictions with live measurements to steer the network toward configurations that meet SLOs.

4

PREDICTION-GUIDED CONTROL

4.1 INTRODUCTION

The previous chapter presented `Parsimon`, a fast, approximate simulator for predicting tail latency in data center networks; in §3.6, we also introduced `m3`, an ML-based alternative that is faster and more accurate on real deployments. Together, these predictors enable rapid "what-if" exploration: given a workload and topology, they can quickly estimate how tail latency would change under different configurations. This chapter builds on these predictors to address a broader question: how can we use approximate prediction to automatically tune network configurations in closed loop?

Achieving class-specific tail latency SLOs in multi-class data center networks is challenging. Tail latency depends on a complex interplay of factors: the workload itself (request rates, flow size distributions, burstiness), the network topology and its capacity, the congestion control algorithm, load balancing decisions and potential imbalances across paths, and end-host behaviors. As discussed in Chapter 2, modern networks expose a broad control surface—end-host congestion control parameters, switch queueing policies, scheduling weights, and more—but these parameters are typically configured once and adjusted only in rare circumstances. Yet their effects on tail latency can be profound: small changes can reshape latency distributions across multiple traffic classes. Further, the appropriate settings are highly workload dependent and time varying. Burstiness, correlated behavior, and interactions between traffic classes affect outcomes in ways that are hard to predict from first principles.

One approach to finding good configurations is model-free autotuning. Systems like Microsoft's SelfTune [52, 81] tune a configuration to a scalar reward signal through online trial-and-error. However, these methods do not explicitly model workload state and lack a natural mechanism for enforcing class-specific tail-latency SLOs under non-stationary traffic. Moreover, they can struggle with high-dimensional, non-convex configuration spaces, especially when reward signals are noisy. Our experiments show that SelfTune fails to converge within practical time budgets for tail-latency-sensitive workloads.

This chapter presents `Polyphony`, a prediction-guided controller that uses fast approximate models like `Parsimon` and `m3` to guide network configuration in closed loop. `Polyphony` operates on a timescale of minutes, with the goal of anticipating future congestion and reconfiguring the network to avoid it. Our approach is complementary to prior

methods like congestion control and traffic engineering; for example, `Polyphony` can adapt congestion control parameters to meet SLOs as workloads evolve.

The key insight is to treat predictions as priors rather than perfect oracles. As we will see in §4.2.1, predictors like `Parsimon` and `m3` have significant error, so naïvely following their recommendations leads to suboptimal configurations. Instead, `Polyphony` corrects for model error by learning a residual function online using a Gaussian process surrogate, which captures the difference between predicted and observed performance. This requires measuring both the workload—which the predictors take as input—and the resulting tail latency. `Polyphony` gathers workload information using eBPF probes that capture flow-level traffic and aggregate it hierarchically across hosts.

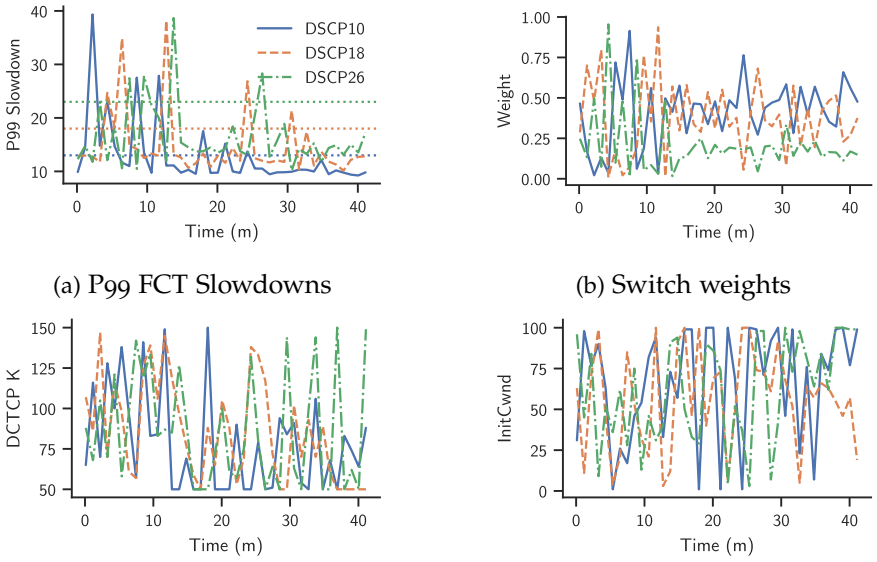
Closing the loop on a live system introduces additional concerns. Tail latency measurements are inherently noisy, especially at extreme percentiles, so `Polyphony` applies adaptive denoising to smooth variance across trials and avoid reacting to spurious fluctuations. To handle workload shifts, `Polyphony` monitors for regime changes and resets its model when traffic patterns change significantly. Finally, because exploratory actions risk degrading performance, `Polyphony` constrains its search to a trust region centered on the best known configuration.

We evaluate `Polyphony` on CloudLab using both `Parsimon` and `m3` as predictors, tuning nine network parameters across three traffic classes. Starting from a configuration that violates class-specific tail latency SLOs, `Polyphony` with `m3` converges to meet all SLOs within approximately ten minutes, while SelfTune makes little progress even after sixty minutes. When we introduce large periodic workload shifts, `Polyphony` adapts and re-stabilizes within fifteen minutes, while SelfTune shows no clear signs of adaptation. We also evaluate `Polyphony` with `m3` on larger-scale ns-3 simulations and conduct ablation studies demonstrating that residual modeling, trust regions, and denoising are each necessary for efficient convergence.

4.2 BACKGROUND AND MOTIVATION

The setting for our work is a network where traffic is divided into multiple service classes, each with its own tail latency objective. For example, an operator might require that 99% of flows in one class must have a flow completion time (FCT) slowdown no greater than 10×, while another class may be able to tolerate a slowdown up to 20×. Following Mogul and Wilkes [65], we call this target a service level objective (SLO), and the measured performance metric (P99 FCT slowdown) a service level indicator (SLI). The bound (e.g., 10 or 20×) is the SLO threshold.

Our approach is agnostic to the unit of measurement, but to be concrete we assume it is a remote procedure call (RPC), remote memory



(a) P99 FCT Slowdowns

(b) Switch weights

(c) DCTCP K

(d) Init CWND

Figure 4.1: Model-free Bayesian optimization leads to poor behavior during convergence. Fig. 4.1a shows the P99 FCT slowdown of three traffic classes over time. Horizontal lines indicate SLO thresholds. Both configurations and tail FCT slowdowns oscillate as the optimizer explores the parameter space.

operation (RDMA), or independent data transfer. Latency is measured as the time to complete the transfer, including the transmission, propagation, and queueing delay, from when the first packet is available to be sent until the last packet arrives at the destination. We include in the latency any queueing at the end host needed for traffic shaping or congestion control. Following industry practice, we modify the host operating system to sample tail latencies as input to our control system.

As in Chapter 3, we use FCT slowdown as our performance metric. `Polyphony` supports performance targets to be defined separately for different message sizes, e.g., to ensure that medium-sized messages aren't starved. In this chapter, we focus on aggregate tail performance.

We assume all network hosts and switches are configured identically, but in a class-aware manner. Each traffic class is assigned its own dedicated queue at each network link; each class is given a scheduling weight that controls how frequently packets from that class are scheduled when other traffic is present. All hosts and switches implement the same congestion control algorithm, but the parameters controlling the behavior of the algorithm, such as the initial window size or its overall aggressiveness, are traffic class specific. The prediction models we use generalize across many commonly used congestion control algorithms [56, 95], but for our proof of concept we narrow our focus to DCTCP [6]. It contains an initial congestion window and a threshold

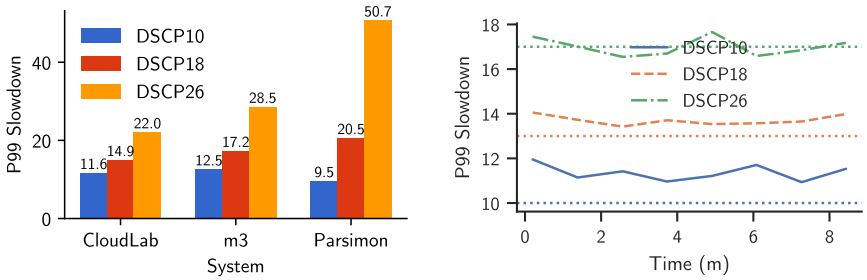


Figure 4.2: Model error and its consequences for three classes with varying SLOs. `m3` is trained on CloudLab while Parsimon uses packet-level simulation. (a) Even `m3` mispredicts CloudLab's p99 FCT slowdown by 18% on average. (b) When the best configuration predicted by `m3` is applied to the network, all three traffic classes violate their SLOs, even though they can all be met simultaneously. The configuration is kept the same across samples; variance is due to noise.

parameter, K, for switch queues to trigger end hosts to reduce their sending rate; these parameters are allowed to vary by traffic class.

Whether a specific traffic class meets its objective depends on the specific workload and parameters of that class, but also that of other classes. Class-based switch scheduling allocates a worst case minimum share of the line rate to each class in proportion to its weight, but scheduling is also work conserving. If a class does not have traffic present (e.g., because it is bursty) the unused capacity is proportionately split among the other traffic classes with traffic present. As a result, the effect of individual changes may not be monotonic. For example, the DCTCP K value controls the target amount of queueing at congested switches. Reducing this value can improve tail latency for flows that fit within the initial congestion window, but it can also hurt tail latency by reducing the ability of the traffic class to take advantage of the idle periods of other classes.

4.2.1 *Existing methods*

Given that the network exposes parameters for control, our goal is not a "best configuration" but continuous regulation: adjusting settings in closed loop to track SLOs as workloads evolve. When operating conditions change—for example a shift in traffic patterns or workload intensity—the controller should adapt while minimizing SLO violations. We therefore evaluate approaches by: (i) convergence time (time to reach per-class SLO compliance), (ii) regret (cumulative deviation from compliance during adaptation), and (iii) fairness (whether deviations are borne evenly across classes). In our experiments, we treat the SLOs for each traffic classes as equally important, but `Polyphony` can also prioritize certain classes with weighted objectives.

MODEL-FREE AUTOTUNERS. A prominent line of prior work adjusts parameters using black-box autotuning. These methods are model-free: they iteratively deploy settings and observe outcomes on the live system. For stability, they typically operate on long time horizons, with each iteration lasting up to hours or days. A common example is Bayesian optimization [21], which selects configurations to test by balancing exploration (trying uncertain settings) and exploitation (choosing promising ones). While model-free methods can eventually find good configurations, they must search by running real experiments on the target system, risking SLO violations (regret) in the process. Fig. 4.1 shows standard Bayesian optimization tuning a small CloudLab network with three traffic classes and nine parameters (per-class switch weight, DCTCP marking threshold, and initial congestion window). Although the optimization eventually settles after about thirty minutes, it first explores widely, causing users of the system to experience significant SLO violations. We would expect these oscillations to be more pronounced with more traffic classes and additional per-class parameters.

An emerging class of model-free autotuners uses reinforcement learning (RL). RL tuners like SelfTune [52] and OPPerTune [81] update parameters by acting in discrete rounds: in each round, the tuner picks a configuration, the system runs for some time—typically ranging from an hour to days—and a scalar reward is fed back. These approaches rely on round-level measurements being information-rich and averaging out noise. In our setting with short timescales and noisy tail latency measurements, these conditions do not hold. As a result, reward signals can be noisy and gradients weak, causing updates to shrink and learning to stall. We apply SelfTune to our setting in the evaluation (§4.9). We find that, whether by Bayesian optimization or reinforcement learning, model-free autotuning struggles to track tail latency SLOs on short timescales in dynamic, noisy environments.

EMERGING FAST MODELS. Meanwhile, recent work like `Parsimon` and `m3` have developed fast, approximate simulation methods for predicting tail latency in data center networks. One idea would be to use their predictions to drive configurations directly in open loop. Unfortunately, these models have significant error. Both rely on approximations—such as decomposing networks into independent links or paths—that trade accuracy for speed, and `Parsimon` omits end-host effects like OS scheduling and application behavior. ML models like `m3` are also sensitive to distribution shift: the conditions under which they are trained may not perfectly match deployment, or may change over time. Fig. 4.2a shows an example: even `m3`, which is trained directly on the target platform, exhibits an average of about 18% error in predicting 99th percentile FCT slowdown across traffic classes. Fig. 4.2b shows what happens when we use the model to

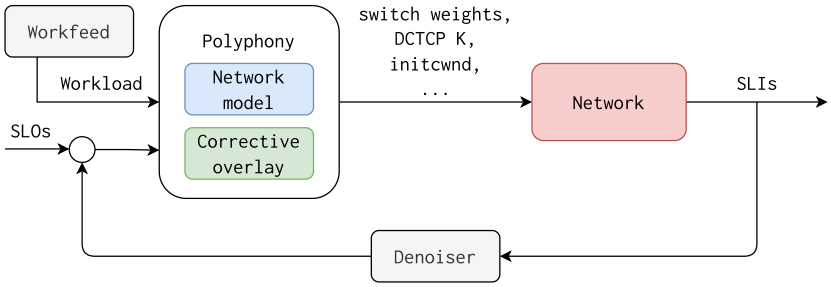


Figure 4.3: Polyphony's closed-loop architecture.

find its predicted best configuration, and then apply it directly. Even though the SLOs are achievable, the deployed configuration misses all SLOs.

Model-free search learns from live measurements but can incur SLO violations due to wide exploration; model-based optimization can avoid unsafe exploration but inherits the model's errors. Neither alone provides fast, low-regret adaptation for tail latency SLOs. Polyphony combines aspects of both approaches by coupling a fast model with online corrective feedback. The next section outlines Polyphony's closed-loop architecture and introduces how it blends simulation and live measurement to achieve SLO compliance.

4.3 POLYPHONY OVERVIEW

Given a live network that serves multiple traffic classes with per-class tail latency SLOs, our goal is to continuously adjust a set of knobs—like switch weights or congestion control parameters—so all SLOs are met and tail latency is minimized. As a secondary goal, allocations should be *fair* in the sense that no class is persistently disadvantaged relative to its SLO. We frame this as a regulation problem: Polyphony acts as an online controller that tracks per-class SLO constraints via small, bounded adjustments.

Fig. 4.3 depicts Polyphony's closed-loop architecture. The core idea is to combine a cheap, approximate performance model with an online-learned corrective overlay. Polyphony then uses the overlay to compute the next bounded adjustment within a trust region for a network setting to try next. By optimizing a predictive model rather than the live system, Polyphony reduces the number of costly system measurements required to find good configurations, and limits the risk of applying undesirable settings. The system takes as input the target SLOs and the current service-level indicators (SLIs), and it outputs the next setting to apply. The performance models require workload information as input. To measure workloads, Polyphony bundles a lightweight Workfeed component that uses eBPF to capture flow events and reconstruct flow sequences for the models.

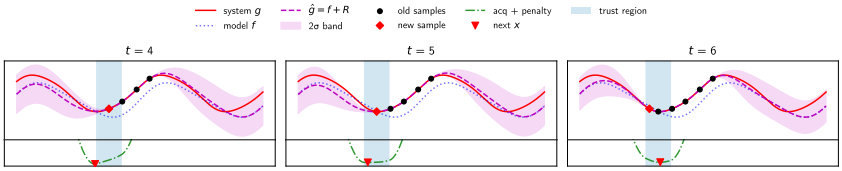


Figure 4.4: An illustration of Bayesian optimization with a residual surrogate and a trust region. Each panel shows one control iteration ($t = 4, 5, 6$). The acquisition function (bottom curve) is quadratically penalized outside the trust region. Over successive steps the surrogate improves near the new sample and the trust region recenters, guiding the optimizer toward better configurations while avoiding unsafe exploration.

At each control interval, `Polyphony` computes a bounded update to the control knobs. First, it collects SLIs, including per-class tail slowdowns. These metrics capture the performance seen by each class and serve as feedback for updating the corrective overlay. Next, it filters these measurements to reduce the effects of noise and transient spikes, producing a smoothed estimate of the system performance. Using this filtered signal, `Polyphony` updates its corrective overlay. This model captures the discrepancy (i.e., residual) between the predictions of the fast, approximate simulator and the live measurements from the real system. We use the estimated residual to improve the predictive accuracy of the overall model. Then, `Polyphony` searches for a set of network knobs within a trust region that maximize the expected performance improvement.This step balances exploration with caution, ensuring that new settings are only tried if the model predicts an improvement with high confidence. Finally, the selected configuration is sent to the switches and hosts to actuate the intervention.

The following sections describe `Polyphony`'s methods in detail. First, we focus on settings with stable workloads in §4.4. Then, we describe adaptation to workload changes in §4.5.

4.4 SAFE PREDICTION-GUIDED OPTIMIZATION

To start, we view the network as a black box that, for a workload, accepts a vector of parameters $\mathbf{x}$—switch weights, marking thresholds, etc.—and emits a vector of SLIs $\mathbf{y}$, such as per-class tail latencies. Throughout this section, we assume the workload is fixed and stable, and drop it from notations to avoid clutter. Alongside the network, we have an approximate model, like `Parsimon` [95] or `m3` [56], that takes the same $\mathbf{x}$ and produces *predicted* SLIs for an independent sample of the same workload. We assume that the model makes errors but is cheap to evaluate, and that the real system is noisy and risky to probe. `Polyphony`'s principal idea is to learn the difference between them and then optimize the performance using a *corrected* version of

the model. It also optimizes within a safety envelope to avoid straying from known good configurations. To make these ideas more precise, we begin by introducing some notation.

4.4.1 *Definitions and problem formulation*

We first define the parameter space. `Polyphony` considers two kinds of parameters. The first lie in a hypercube, where each parameter varies independently. Examples of these are congestion control parameters or initial congestion windows. The second lie in a simplex: they are constrained to be non-negative and sum to one. These represent allocations or proportions, such as switch weights which must be distributed among traffic classes. Polyphony manipulates a vector

$$\mathbf{x} \;=\; \big(\, h^{(1)},\, h^{(2)},\, \ldots,\, h^{(M_h)},\; u^{(1)},\, u^{(2)},\, \ldots,\, u^{(M_u)} \big),$$

where $h^{(k)} \in [0,1]^{d_k}$ are hypercube parameters and $u^{(\ell)} \in \Delta^{d_\ell}$ are simplex parameters. Combining these domains, the full parameter space is

$$\mathcal{X} \;=\; \Big(\prod_k [0,1]^{d_k}\Big) \times \Big(\prod_\ell \Delta^{d_\ell}\Big).$$

All coordinates are scaled to $[0,1]$ for optimization and inverse-normalized for actuation.

For the particular workload, we think of the system as a function $s(\mathbf{x}) : \mathcal{X} \to \mathbb{R}^N$ which maps a vector of parameters to a vector of SLIs. The model $m(\mathbf{x}) : \mathcal{X} \to \mathbb{R}^N$ is defined similarly for the same workload. Although $s(\mathbf{x})$ and $m(\mathbf{x})$ each return a N-dimensional vector of SLIs, an optimizer needs a total order over configurations to decide whether one setting is better than another. `Polyphony` therefore collapses the vector into a single scalar cost J. Doing so has two main advantages: 1) it reduces the problem to scalar optimization with a clear improvement signal, and 2) it allows us to express SLO compliance and fairness as a single quantity to optimize instead of exploring a Pareto frontier.

To guide optimization, we need a scalar objective that reflects our goals of SLO compliance and fairness. Let $\mathbf{y} = (y_1, \ldots, y_N) \in \mathbb{R}^N$ be the observed SLIs (e.g., tail latencies) for each of the N traffic classes, and let $\mathbf{v} = (v_1, \ldots, v_N) \in \mathbb{R}^N$ be their corresponding SLOs thresholds. To ensure SLO compliance, we want each class's observed slowdown y_i to be less than its threshold v_i, so we define the per-class ratio $r_i = y_i / v_i$. A value $r_i \leq 1$ indicates compliance. Optimizing for SLO compliance then becomes mimizing some aggregate of these ratios across classes.

We choose the maximum as our aggregate because SLO violations are typically unacceptable even if they affect only a single class. Using the maximum of the ratios prioritizes reducing the worst violation. However, maximum is non-differentiable, which prevents the use of

gradient-based optimization methods. To address this, we use LogSumExp (LSE) [17, Sec. 3.1.5] as a smooth approximation of maximum, with sharpness controlled by parameter $c > 0$:

$$\mathrm{LSE}(r_1, \ldots, r_N) \;=\; \frac{1}{c} \ln\Big(\sum_{i=1}^{N} \exp(c\, r_i) \Big),$$

As $c \to \infty$, the function approaches a sharp maximum, while smaller values of c yield a smoother behavior.

To encourage fair allocations, we add a fairness penalty which tries to equalize the r_i ratios. Let $\bar{r} \;=\; \frac{1}{N} \sum_{i=1}^{N} r_i$ be the mean of the ratios. We define

$$\mathrm{fairness}(r_1, \ldots, r_N) \;=\; \frac{1}{N} \sum_{i=1}^{N} \big|\, r_i - \bar{r} \,\big|.$$

The final objective is then

$$J(r_1, \ldots, r_N) \;=\; \mathrm{LSE}(r_1, \ldots, r_N) \;+\; \lambda\, \mathrm{fairness}(r_1, \ldots, r_N),$$

where $\lambda \geq 0$ is a trade-off constant. Lower values of J indicate a smaller worst-case slowdown ratio and/or a fairer spread of ratios across classes, depending on the weight λ.

Recall that for a particular workload, the system $s(\mathbf{x})$ and model $m(\mathbf{x})$ each return a vector of SLIs given a configuration $\mathbf{x}$. To calculate r_i ratios used for objective (J) computation, we can use $s(\mathbf{x})$ or $m(\mathbf{x})$. We call the former $g(\mathbf{x})$ and the latter $f(\mathbf{x})$. In words, $g(\mathbf{x})$ is the system's observed performance and $f(\mathbf{x})$ is the model's predicted performance. In what follows, we refer to g as "the system" and f as "the model."

4.4.2 *Modeling residuals*

The controller must minimize a system g that is noisy and risky to sample while taking advantage of a model f that could have significant modeling error. Directly learning $g(\mathbf{x})$ is difficult because it can be highly nonlinear and discontinuous. Instead, `Polyphony` learns the *residual*

$$R(\mathbf{x}) \;=\; g(\mathbf{x}) - f(\mathbf{x}),$$

which is typically smoother and smaller in magnitude.

GAUSSIAN-PROCESS SURROGATE. We treat the residual as a random function drawn from a GP $R \sim \mathrm{GP}$. Given a set of residual observations $\{(\mathbf{x}_j, R_j)\}$, standard GP inference returns a posterior mean $\mu_R(\mathbf{x})$ and variance $\sigma_R^2(\mathbf{x})$.We use $\hat{g}(\mathbf{x}) = f(\mathbf{x}) + \mu_R(\mathbf{x})$ as the bias-corrected predictor and $\sigma_{\hat{g}}^2(\mathbf{x}) = \sigma_R^2(\mathbf{x})$ as its uncertainty. This uncertainty estimate provides the risk budget that supports the safe optimization strategy that follows.

4.4.3 *Bayesian optimization over the surrogate*

Our surrogate $\hat{g}$ supplies both predictions and calibrated uncertainty. This leads naturally to the use of Bayesian optimization to balance exploration and exploitation.

Bayesian optimization (BO) iteratively chooses the next configuration by maximizing an *acquisition function* built on top of a fast *surrogate* of the true objective. In our case

$$\text{surrogate} \;=\; \hat{g}(\mathbf{x}) \quad \text{from §4.4.2},$$

and the acquisition is Expected Improvement (EI).

EXPECTED IMPROVEMENT. Let $g_{\text{best}} = \min_{j<t} g(\mathbf{x}_j)$ be the smallest *observed* objective after $t-1$ iterations. For a candidate $\mathbf{x}$ with surrogate mean $\mu_{\hat{g}}$ and standard deviation $\sigma_{\hat{g}}$,

$$\text{EI}(\mathbf{x}) \;=\; \big(g_{\text{best}} - \mu_{\hat{g}}\big)\,\Phi(z) + \sigma_{\hat{g}}\,\phi(z), \qquad z = \frac{g_{\text{best}} - \mu_{\hat{g}}}{\sigma_{\hat{g}}},$$

where $\Phi(\cdot)$ and $\phi(\cdot)$ are the standard normal CDF and PDF. EI is the *expected drop in the objective* if we were to sample at $\mathbf{x}$. Anchoring EI to the smallest *measured* value guards against model over-optimism: a point can only show positive improvement if it is expected to beat what the system has actually achieved, not merely what the model predicts. Because `Polyphony` is a minimizer, the implementation minimizes $A(\mathbf{x}) = -\text{EI}(\mathbf{x})$:

$$\mathbf{x}_t \;=\; \arg\min_{\mathbf{x}\in\mathcal{X}} \; A(\mathbf{x}).$$

EI balances exploitation of low $\mu_{\hat{g}}$ with exploration in regions of high $\sigma_{\hat{g}}$, aiming for sample-efficient improvement over time.

HANDLING SIMPLEXES. Bayesian optimization typically operates in an unconstrained, Euclidean space. To accommodate simplex-valued parameters—such as traffic class weights that must be non-negative and sum to one—we optimize in the unconstrained pre-image and map the result through a softmax transformation. This allows gradient-based methods to search freely while ensuring that the final parameter vector lies in the simplex. The surrogate and acquisition functions are defined over the unconstrained space but always evaluated on the mapped simplex point.

4.4.4 *Safe exploration*

While Bayesian optimization efficiently explores the parameter space $\mathcal{X}$ by balancing exploration and exploitation, it does not guarantee safety. Probing g can be risky: unconstrained exploration might lead the optimizer to query points $\mathbf{x}$ that violate SLOs, even if the surrogate $\hat{g}$ predicts otherwise.

TRUST REGION. To mitigate this risk, `Polyphony` uses a safe exploration strategy that builds on well-understood methods from the optimization literature [30]. Instead of optimizing the acquisition over the entire parameter space $\mathcal{X}$, we restrict the search to a *trust region* centered around the *best configuration* observed so far. Let $\mathbf{x}_{\text{best}}^{(t)} = \arg\min_{\mathbf{x}_j, j<t} g(\mathbf{x}_j)$ be the parameters corresponding to the best *observed* system performance $g_{\text{best}} = g(\mathbf{x}_{\text{best}}^{(t)})$ after $t-1$ iterations. The optimizer at step t focuses its search within a neighborhood of $\mathbf{x}_{\text{best}}^{(t)}$.

DISTANCE METRIC. Defining this neighborhood requires a distance metric on the mixed parameter space $\mathcal{X}$. Each configuration $\mathbf{x} \in \mathcal{X}$ consists of 1) a hypercube part $\mathbf{h} \in \prod_{k=1}^{M_h} [0,1]^{d_k}$, and 2) M_u simplex blocks $u^{(1)}, \ldots, u^{(M_u)} \in \Delta^{d_\ell}$. We measure separation with

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\underbrace{\|\mathbf{h}_\mathbf{x} - \mathbf{h}_\mathbf{y}\|_2^2}_{\text{hypercube}} + \alpha \sum_{\ell=1}^{M_u} \underbrace{d_A\big(u_\mathbf{x}^{(\ell)}, u_\mathbf{y}^{(\ell)}\big)^2}_{\text{simplex}}},$$

where d_A is the Aitchison distance commonly used for compositional data [1] and $\alpha \geq 0$ balances the Euclidean and compositional parts. The weight α is tuned so that the simplex and hypercube terms contribute on the same numerical scale.

ENFORCEMENT. Enforcing a hard constraint $d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon$ during optimization can be complex, especially for simplex parameters, which we optimize in an unconstrained space. `Polyphony` enforces the trust region by adding a penalty term to the acquisition function:

$$\text{penalty}(\mathbf{x}\,;\, \mathbf{x}_{\text{best}}^{(t)}, \epsilon) = \begin{cases} 0 & d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon, \\ \beta \cdot \big(\, d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) - \epsilon \,\big)^2 & \text{otherwise,} \end{cases}$$

where ϵ is the current trust-region radius and $\beta > 0$ controls the penalty severity. The optimizer minimizes

$$A_{\text{TR}}(x) = -\,\text{EI}(x) + \text{penalty}(\mathbf{x}\,;\, \mathbf{x}_{\text{best}}^{(t)}, \epsilon)$$

thereby searching freely *inside* the ball $d(\mathbf{x}, \mathbf{x}_{\text{best}}^{(t)}) \leq \epsilon$ while rapidly discouraging excursions beyond it. Because the penalty is quadratic and continuous, gradients remain well-behaved. In our experiments, we set $\beta = 10^4$.

Fig. 4.4 illustrates the concepts we have introduced so far. Next, we describe how `Polyphony` adapts to new operating regimes, and how it handles data to remain both reactive and stable over long time horizons.

4.5 ADAPTATION AND DATA QUALITY

Thus far, we have described methods that allow `Polyphony` to converge to good solutions so long as g remains stationary. Real networks, however, can experience step changes: traffic mixes shift, hot spots appear. To adapt to new regimes, `Polyphony` must 1) detect that the regime has changed, and 2) forget stale data that would mislead the GP. This section discusses these mechanisms. In addition, we describe methods to ensure `Polyphony` remains effective over long timescales.

4.5.1 *Adapting to changing g*

Consider how `Polyphony`—as currently described—would respond if the network suddenly experienced a disruption, such as a sudden influx of traffic. The value of our objective g would spike, prompting the controller to adjust parameters to achieve SLO conformance. However, the trust region would still be centered around the old best point, tying the optimization to that region and causing it not to make progress. To prevent this, we first maintain a *rolling* best objective rather than a static one:

$$g_{\text{best}}(t) \;=\; \min_{j=t-W_g}^{t-1} g(\mathbf{x}_j),$$

where W_g is the size of the rolling window. Next, we implement a simple scheme for objective spike detection and trust region reset. If the current measurement exceeds the rolling best objective by a factor of ϕ for k consecutive iterations, $g(\mathbf{x}_t) \;>\; \phi\, g_{\text{best}}(t)$, we treat the event as a regime shift, and we 1) reset the g_{best} window and 2) designate the current configuration as $\mathbf{x}_{\text{best}}^{(t)}$.

4.5.2 *Processing data samples*

ROLLING DATA. As `Polyphony` runs over time, the number of samples it collects can grow without bound. This behavior is undesirable for two reasons. First, GP inference scales as $\mathcal{O}(n^3)$, where n is the number of samples. Second, old points from outdated regimes can dominate, forcing the surrogate to confidently explain behavior that is no longer relevant, making adaptation sluggish. To eliminate this effect, we maintain a rolling window of samples of size W_s.

FILTERING DATA. As `Polyphony` converges to a good solution, every new sample is likely to lie in the same small neighborhood, and thus nearly identical in the input space. Updating a GP requires inverting a covariance matrix, which can become ill-conditioned if most of its rows are almost the same. This can cause the GP posterior mean and variance to become unreliable. `Polyphony` employs a simple data

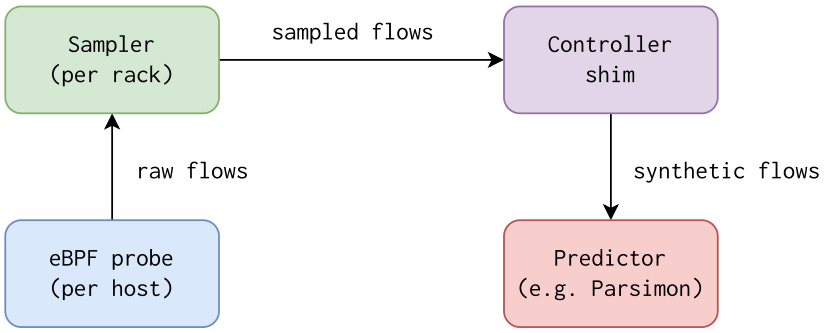


Figure 4.5: Proposed pipeline for workload collection.

admission policy to mitigate this risk. We say a configuration is *novel* if it is far enough away from all other configurations in the rolling window of samples, and we say a sample is *surprising* if the GP mispredicts its outcome. `Polyphony` admits a sample if at least one of these conditions is met. This rule acts as a lightweight guard that keeps the GP numerically stable.

DENOISING DATA. Real systems are noisy, especially when measuring metrics like tail latency. Feeding jittery data into the GP can lead it to interpret noise as signal, slowing learning, and chasing spurious spikes. Before updating the GP with a new sample, `Polyphony` applies a causal low-pass Gaussian smoother whose width adapts to the volatility of the signal. If the signal becomes completely stable, the filter automatically turns off. We evaluate the effect of the denoiser in §4.9.4.

4.6 MEASURING WORKLOADS ONLINE

Recall that the performance models used by `Polyphony` require workload estimates—representative sequences of flows with their sizes, classes, and arrival times—as input. In a live network, obtaining these estimates requires capturing flow-level metrics from production traffic. To do this, we built `Workfeed`, a lightweight distributed system that monitors ongoing traffic and provides workload traces for model evaluation. `Workfeed` measures TCP flows to mirror the workload generator; extending it to capture RPC-level traces over long-lived connections is future work.

4.6.1 *Overview*

The main idea is to record flow-level events via lightweight eBPF probes at each host and then use them to reconstruct workload traces for downstream modeling. To reduce fan-in at the controller, we employ a hierarchical aggregation strategy, where each host sends its

records to a rack-level aggregator, which in turn pools the data and forwards it to the controller in batches. However, even with hierarchical aggregation, large deployments can generate substantial data volume that overwhelms a single controller. To address this, we apply sampling at the rack level. Our performance models require accurate workload statistics, not perfect per-flow tracking. Large flows, though rare, carry most of the bytes and dominate queue buildup; missing even one can distort tail latency predictions. Small flows are numerous but individually contribute little to congestion, so we can subsample them heavily and still reconstruct accurate aggregate statistics through interpolation. This size-aware sampling strategy reduces data volume while preserving the distributional properties needed for modeling. Fig. 4.5 depicts the pipeline.

- *At each host*, we attach eBPF probes to TCP socket setup and teardown events. These collect one record per TCP connection, including the source and destination, DSCP marking, byte count, and start and end timestamps.
- *At each rack*, a small user-space program receives these flow records and applies a deterministic sampling function. The goal is to keep all large flows, a smaller fraction of medium flows, and a sliver of small flows, while ensuring each DSCP class has enough data.
- *At the controller*, a shim expands each sampled flow back into many virtual flows using Poisson resampling and exponential jittering. Each replica is assigned a unique identifier to enable ECMP path diversity in simulation. The resulting list of flows is a statistical approximation of the original unsampled traffic, and it is presented in the format the simulators expect.

Subsequent sections describe these steps in more detail.

4.6.2 *Per-host probing*

Each host loads a small eBPF program that hooks into TCP socket lifecycle events. eBPF is a lightweight, safe, in-kernel virtual machine for monitoring and control tasks. It allows us to insert custom logic directly into the TCP stack without kernel patches or kernel crashes.

We use two specific tracepoints to capture complete flow information. The `inet_sock_set_state` tracepoint fires on TCP state transitions; we use it to detect when connections enter the ESTABLISHED state and record their start timestamp. The `tcp_destroy_sock` tracepoint fires when the kernel tears down the socket; at this point we capture the final byte counts, extract the DSCP value from the TOS field, and compute the flow duration. This approach ensures we collect complete flow statistics without missing any bytes from the connection

lifecycle. Completed flow records are pushed to a zero-copy kernel ring buffer (16 MB). A user-space process running on each host continuously polls the ring buffer, pulling flow records in batches. The same process accumulates up to 128 flows or waits up to 200 ms before packaging them into a UDP datagram and sending them to the per-rack sampler. This design avoids locks and imposes minimal performance impact while providing exact per-flow data. We intentionally do not apply sampling at the host level: the eBPF probes are lightweight enough that capturing every flow introduces negligible overhead, and deferring sampling to the rack level allows for better control over size-stratified rates.

4.6.3 *Per-rack sampling*

To keep the controller from being overwhelmed by every TCP flow, especially in very large networks, we support sampling at the rack level. The key observation is that not all flows are equally important for workload modeling: large flows, though rare, carry most of the bytes and drive queue buildup, while small flows are numerous but individually contribute little to congestion. We therefore use size-aware sampling with configurable per-bucket rates. In a typical configuration, we retain all large flows (e. g., > 1 MB), keep a moderate fraction of medium flows, and sample small flows more aggressively. This approach preserves the tail accuracy of workload models—if we instead sampled uniformly at, say, $1/32$, many observation windows would contain zero large flows, distorting queue occupancy predictions.

The sampling decision is made using a deterministic hash of the flow's 5-tuple and DSCP value. For each flow, we compute a 64-bit hash and normalize it to $[0, 1)$; the flow is sampled if this value falls below the configured rate for its size bucket. This approach ensures that a given flow is either always sampled or always dropped, regardless of which host observes it or when it is measured, enabling consistent treatment across the distributed system. Each sampled flow is tagged with a weight $w = 1/\text{rate}$ that indicates how many flows it represents (e. g., a flow sampled at rate 0.03125 receives weight 32). Sampled flows are grouped into batches and sent to the controller over TCP with a short timeout; if the controller is unavailable, flows are dropped on a best-effort basis.

4.6.4 *Expanding samples into workloads*

At the controller, we expand the sampled flows into a full synthetic trace. The controller runs this process once per control epoch. The sampled flows arrive tagged with their weights, and we sort them by start time. Our approach employs similar statistical techniques to prior work on synthetic trace generation [46], using Poisson resampling to

construct workloads from data. For each sampled record (except the last), we do the following:

1. Emit the original flow at its exact timestamp.
2. Draw a replica count from a Poisson distribution based on the flow's weight. Poisson sampling adds natural variance so that the trace avoids identically-sized blocks of replicas.
3. Determine the time window until the next sampled flow starts, and emit replicas at jittered timestamps inside this window. We draw inter-arrival gaps from an exponential distribution to introduce realistic temporal clumping while respecting the window boundary.

We do not replicate the last sampled flow in each batch to avoid extrapolating beyond the measurement window. Each expanded flow is assigned a unique identifier so that simulators can model ECMP path diversity without altering the flows' 5-tuples. This procedure preserves the statistical properties of the original workload while reconstructing a plausible fine-grained arrival process.

We discuss the limitations of this sampling-and-re-expansion approach, including the loss of some temporal correlations (e.g., incast), in §4.7.

4.7 LIMITATIONS

This section discusses `Polyphony`'s limitations.

NO FORMAL GUARANTEES. `Polyphony` provides no formal guarantees of convergence to globally optimal configurations. Convergence depends on model quality and the effectiveness of the sampling strategy. The trust region ensures monotone improvement in expectation inside one regime, but global optimality is not guaranteed. This safety-performance tradeoff is typical of local optimization methods, which prioritize safe, incremental improvements over global exploration.

SCALING WITH DIMENSIONALITY. As the number of adjusted knobs increases, so does the sample complexity required for accurate modeling and optimization. While `Polyphony` uses Gaussian Process mixtures [55] to scale to moderate dimensions, they remain inherently limited by the curse of dimensionality. This limits `Polyphony`'s scalability in very high-dimensional control problems without additional techniques like dimensionality reduction or structured modeling.

NO CATEGORICAL KNOBS. `Polyphony` currently only handles continuous parameters; categorical choices are not supported. A straightforward extension is to one-hot encode each category and treat the

resulting binary dimensions as additional continuous inputs. We leave this to future work.

CONTROLLER HYPERPARAMETERS. `Polyphony` introduces a set of hyperparameters, including data window size, trust region size, regime shift detector thresholds, and others. All experiments in this section use a single, fixed setting for each. In practice, we have found that most parameters could be set once and did not require tuning. Some, like the trust region radius ϵ, data window size W_s, and regime shift thresholds ϕ and k may need to be adjusted depending on model quality and traffic volatility.

WORKFEED SAMPLING AND RE-EXPANSION. Our rack-level, size-aware sampling coupled with Poisson-based re-expansion preserves basic workload statistics (flow-size distribution, traffic class mix, and aggregate arrival rates), but it does not capture fine-grained temporal structure. Because we replicate flows independently within time windows, patterns like synchronized bursts from multiple senders to a single receiver (incast), or groups of causally-related flows that arrive together, can be spread out or lost entirely. When these patterns significantly affect queueing behavior, the reconstructed trace may underestimate peak congestion even if total bytes are correct. More sophisticated reconstruction that preserves temporal correlation is future work.

4.8 IMPLEMENTATION

We implement `Polyphony` as a modular controller framework in Rust encompassing workload generation, performance modeling, runtime correction, optimization, and system actuation.

WORKLOAD GENERATION. We develop `emu`, a workload generator that emits TCP flows using configurable workloads (e.g., flow size and inter-arrival distributions). Traffic classes are marked with DSCP values (10, 18, 26), enabling per-class policy control. `emu` uses Prometheus [71] to collect aggregate flow metrics (e.g., p99 FCT slowdown) to feed to the controller.

PERFORMANCE MODELS. We extend both `Parsimon` [95] and `m3` [56] to operate over a 9-dimensional parameter space consisting of switch weights, initial congestion windows, and DCTCP marking thresholds for three traffic classes. `m3` is a machine learning-based predictor trained on data collected from the target deployment environment (CloudLab or ns-3, depending on the experiment) using Facebook's public traces [74]. For Gaussian processes in the corrective overlay,

Variant	Description	Model acc.	Δt
`pol/pmn`	Polyphony/`Parsimon` [95]	Low	~95s
`pol/m3`	Polyphony/`m3` [56]	High	~7s
`SelfTune`	`SelfTune` baseline [52, 81]	—	<1ms

Table 4.1: Controller variants used in the evaluation. Δt is the amount of time it takes to compute the next configuration after receiving feedback from CloudLab.

we use the `egobox-moe` [55] library, automatically selecting the best-fit kernel and mean function using the Bayesian Information Criterion.

SYSTEM ACTUATION. For experimentation in CloudLab [29], we implement actuators for per-class switch weights, DCTCP marking thresholds, and initial congestion windows. The first two are configured via the DellS4048 switch's CLI; the last uses the `ip route` utility. We also add support for tuning these parameters in ns-3 [98] by adapting the HPCC codebase [58].

4.9 EVALUATION

To evaluate `Polyphony`'s (`pol`) performance, we ask:

- Does `Polyphony` converge to meet SLOs?
- How long does convergence take?
- Can `Polyphony` adapt to changing conditions?
- How does `Polyphony`'s performance depend on the speed and accuracy of its performance model?
- Which of `Polyphony`'s components influences its performance the most?

We use CloudLab [29] to evaluate `Polyphony` on real machines, where performance depends on real transport protocols, NIC hardware, and system-level scheduling noise. This setting exposes the controller to a range of variability present in real deployments, capturing effects that models and simulations omit.

While CloudLab provides a realistic testbed, it is also resource-constrained, especially in the number of bare-metal switches. To test larger topologies with more diverse workloads, we perform a scalability analysis with ns-3 [98] as the ground truth.

4.9.1 *Setup*

CLASSES AND KNOBS. Across experiments, we split traffic into three classes, corresponding to Differentiated Services Code Point (DSCP) values 10, 18, and 26. Each class is assigned its own configuration of three control knobs: switch weight, DCTCP marking threshold K, and initial congestion window (CWND). These knobs span different layers of the stack and together define a 9-dimensional configuration space.

VARIANTS AND BASELINE. Table 4.1 shows two variants of `Polyphony` using different performance models and the baseline. `pol/pmn` uses `Parsimon` [95] as the performance model. This shows how the controller behaves when the performance model is far slower and less accurate. In Fig. 4.2a, we saw that `Parsimon` had 62% average error on CloudLab, and in our controller experiments, we observe that each controller iteration takes a minute and a half. Compared to `pol/pmn`, `pol/m3` uses a much more accurate machine learned model `m3`, which saw 17.6% average error in Fig. 4.2a and reduces the controller iteration time to seven seconds. Lastly, we configure `SelfTune` as a baseline.

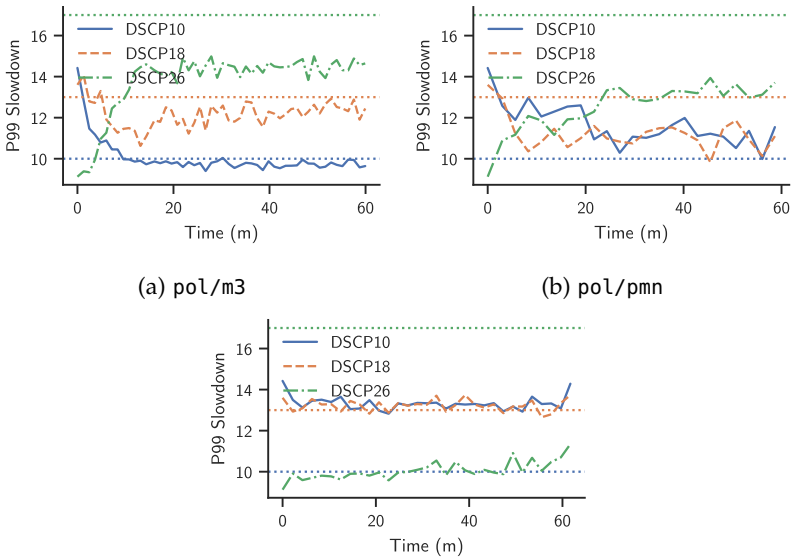
CONFIGURING `selftune`. `SelfTune` has hyperparameters δ and η, which are the perturbation radius and learning rate, respectively. The authors recommend setting $\delta = O(1)$ and $\eta = O(\delta^2)$, so we choose $\delta = 0.1$ and $\eta = 0.01$. To specify a reward function, we reuse our objective from §4.4.1, but we rescale it to be in $(-1, 1)$ using the hyperbolic tangent function. If g is the value of the objective we're trying to minimize, we set the reward to $\tanh(-\beta g)$. The parameter β is tuned to typical values of the objective to prevent the reward from saturating too early or responding too weakly. In our experiments, we set $\beta = 0.3$. To make `SelfTune` tune simplex parameters, we replicate `Polyphony`'s strategy: we tune the parameters in unconstrained space and then apply softmax to map them back to the simplex. Finally, we have found `SelfTune` to be sensitive to noisy measurements. To improve its stability, we wait longer for the objective to settle before computing the reward – two minutes vs. the default one minute.

METRICS. We evaluate each controller variant using four key metrics. First, we track the optimization objective over time, which is a unified metric for SLO compliance and fairness. Second, we report *convergence time*, defined as the first point after which the system meets all SLOs for $k = 3$ consecutive iterations. Third, we measure *hindsight regret*, which integrates the difference between the actual objective and the best achievable in hindsight, penalizing exploration that does not improve the objective. Lastly, we compute *min-max fairness* at the

end of each experiment, defined as the ratio of the smallest to largest normalized slowdown across classes.

CLOUDLAB SETUP. We run our experiments on five `xl170` machines connected via 10G links to a Dell S4048-ON switch. One machine acts as the manager, while the remaining four serve as traffic endpoints: one receiver and three senders. Each machine runs `emu` (§4.8), our workload generator, which issues TCP flows between senders and receiver according to specified traffic patterns. This environment includes real NICs, full transport stacks, and operating system-level variability, allowing us to evaluate the controller under realistic sources of noise and nondeterminism.

4.9.2 *Convergence study*



(a) `pol/m3` (b) `pol/pmn`

(c) `SelfTune`

Figure 4.6: Per-class 99th percentile flow completion time slowdowns under high SLO constraints. `pol/m3` meets all SLOs within a few iterations. `pol/pmn` slowly trends toward SLO conformance but does not meet the SLO for DSCP 10. Finally, `SelfTune` misses SLOs for two of the three classes, showing little improvement when relying on noisy gradient estimates.

We begin by evaluating 1) whether `Polyphony` can discover configurations that satisfy SLOs and 2) how long convergence takes. This test evaluates convergence against a fixed, known workload with three traffic classes and varying levels of SLO stringency. In §4.9.3, we evaluate `Polyphony`'s ability to adapt to dynamically changing workloads from `Workfeed`.

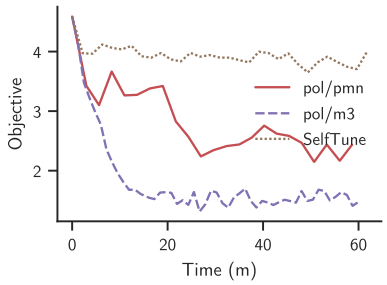
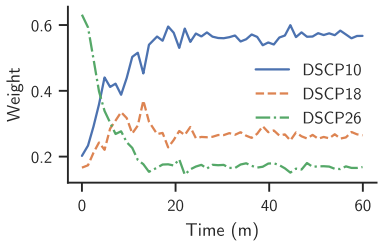
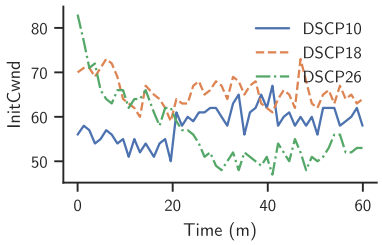


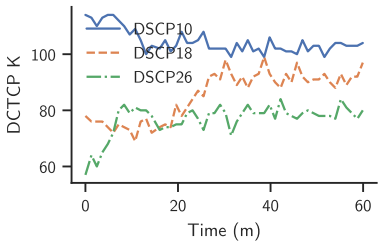
Figure 4.7: Objective over time under high SLO constraints.



(a) Switch weights



(b) Initial CWND



(c) DCTCP K

Figure 4.8: Parameter trajectories for `pol/m3` under tight SLO constraints. `Polyphony` coordinates switch and host tuning to meet SLOs and fairness objectives.

We use publicly released workloads from Meta's data center network [74]. The flow size distributions are sampled from applications like databases, web servers, and Hadoop. Inter-arrival times follow a log-normal distribution with a $\sigma = 2$ for bursty traffic [95]. Flows are evenly distributed across DSCP classes 10, 18, and 26, each contributing 20% for a network load 60%. Experiments run for 60 minutes under three levels of SLO tightness, specified as thresholds for 99th percentile slowdown—one per DSCP: *Low*: (12, 16, 22), *Medium*: (11, 15, 20), *High*: (10, 13, 17).

Table 4.2 summarizes convergence time and objectives across low, medium, and high SLO tightness levels. Fig. 4.6 shows the 99th percentile flow completion time slowdowns for each traffic class over time, under high SLO tightness. `pol/m3` (Fig. 4.6a) converges within 9.5 minutes, satisfying all per-class SLOs. Its behavior remains stable

Variant	Converge Time	Regret	Fairness	Final Obj.
Low SLO tightness				
`pol/m3`	3.7 mins	21.0	0.83	1.49
`pol/pmn`	38.2 mins	67.7	0.66	2.01
`SelfTune`	Not converged	89.7	0.57	3.30
Medium SLO tightness				
`pol/m3`	6.0 mins	20.7	0.86	1.43
`pol/pmn`	Not converged	83.2	0.72	2.09
`SelfTune`	Not converged	127.7	0.46	3.15
High SLO tightness				
`pol/m3`	9.5 mins	24.7	0.89	1.47
`pol/pmn`	Not converged	79.9	0.70	2.44
`SelfTune`	Not converged	154.4	0.47	3.99

Table 4.2: Convergence metrics across SLO tightness levels in CloudLab.

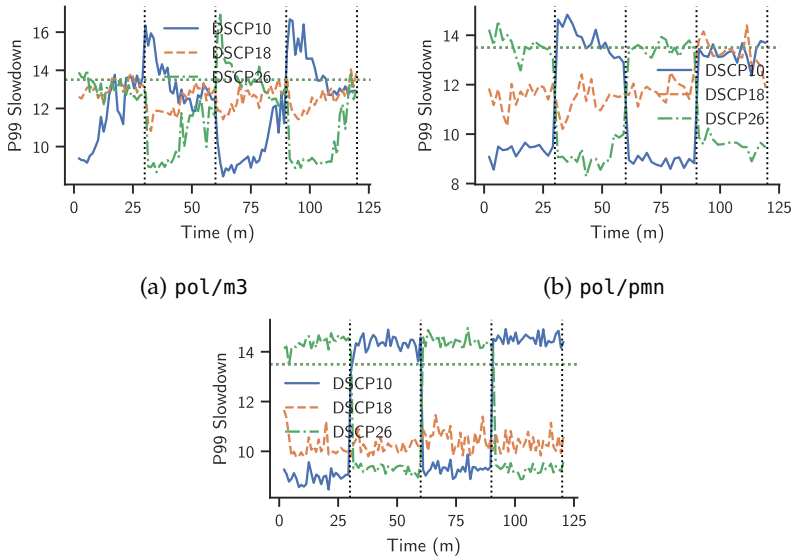
and consistent across traffic classes. In contrast, `pol/pmn` (Fig. 4.6b) converges slower due to a slower and less accurate model. While it stabilizes for DSCP 18 and 26, it fails to meet the SLO for DSCP 10. In Fig. 4.6c, `SelfTune` struggles to meet the SLOs for DSCP 10 and 18, oscillating without converging to better settings over the course of the run. This illustrates the challenge of optimizing under noise and high dimensionality without a performance model to guide search.

Fig. 4.7 presents the global objective evolution under high SLO tightness for three controller variants. `pol/m3` improves quickly and stabilizes in 10 minutes `pol/pmn` achieves gradual objective reduction, but its trajectory flattens early and does not converge within the 60-minute window due to the slower and less accurate model. `SelfTune` improves the objective slightly but struggles to extract a meaningful gradient signal from noisy measurements.

Fig. 4.8 shows the evolution of parameters under `pol/m3`: switch queue weights, initial CWNDs, and DCTCP marking thresholds. All parameters gradually adjust and remain stable over time. We observe that `Polyphony` jointly optimizes host and network settings to balance fairness and performance. Note that the optimal initial window size depends on the scheduling weight, so that the medium traffic class gets the largest window size. A large initial window with high weight will negatively impact the SLOs for other traffic classes.

4.9.3 *Adaptation study*

This section evaluates whether `Polyphony` can adapt its control decisions in the presence of workload changes measured live from `Workfeed`.



(a) `pol/m3` (b) `pol/pmn`

(c) `SelfTune`

Figure 4.9: Per-class 99th percentile flow completion time slowdowns under workload shifts. `pol/m3` adapts within minutes to reestablish SLO compliance. `pol/pmn` converges slowly and has a higher proportion of missed SLOs. `SelfTune` does not meet SLOs and shows no clear adaptation.

Phase	**DSCP 10**	**DSCP 18**	**DSCP 26**
I: 0–30 min	500	2000	3500
II: 30–60 min	3500	2000	500
III: 60–90 min	500	2000	3500
IV: 90–120 min	3500	2000	500

Table 4.3: Traffic load profiles used in the adaptation study. Offered load (in Mbps) varies every 30 minutes to challenge controller responsiveness.

In this experiment, we use the same bursty traffic workloads as described in §4.9.2, and we impose a uniform SLO of 13.5 on all classes. To test adaptability, we introduce three regime shifts during a two hour experiment, and we use `Workfeed` to measure the workload and send it to the controller. Table 4.3 summarizes the offered load profile across four 30-minute phases. Each phase requires the controller to reconfigure both host and switch-level parameters to maintain per-class SLO compliance.

Fig. 4.9 shows the per-class 99th percentile flow slowdowns throughout the adaptation experiment. `pol/m3` (Fig. 4.9a) adapts quickly to workload shifts, reestablishing SLOs and equalizing classes within minutes. `pol/pmn` (Fig. 4.9b) adapts slower, taking nearly a half hour in some cases to reach the SLO threshold. It does not converge fast

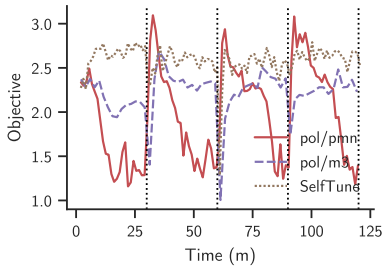
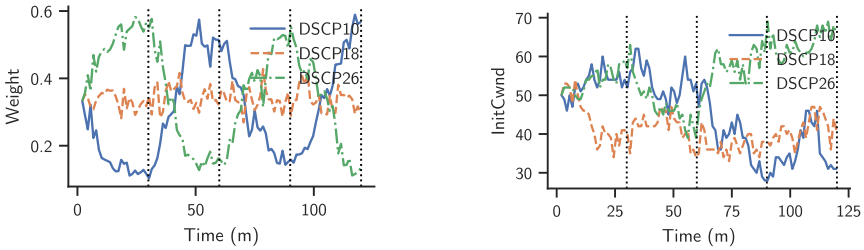
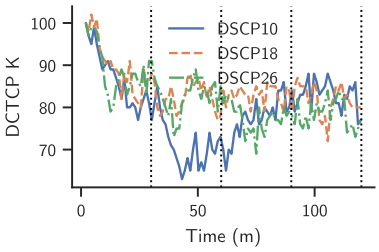


Figure 4.10: Objective over time under regime shifts.



(a) Switch weights

(b) Initial CWND



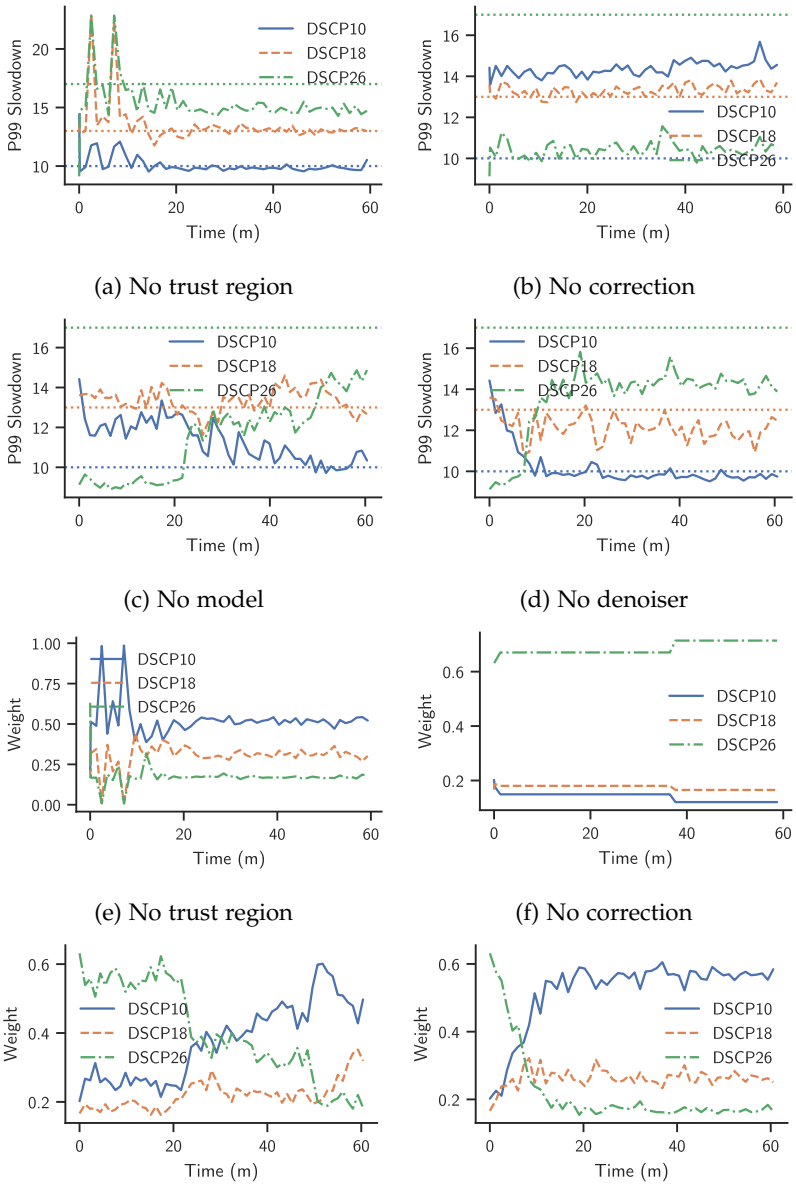
(c) DCTCP K

Figure 4.11: Parameter evolution under `pol/m3`. `Polyphony` gradually updates parameters in response to workload changes.

enough to equalize classes within any half hour window. `SelfTune` (Fig. 4.9c) does little to improve performance and does not respond to workload changes.

Fig. 4.10 shows the global objective over time. `pol/m3` quickly reduces the objective quickly after each workload shift. While `pol/pmn` steers away from large SLO violations, it scores poorly in the fairness dimension, so the objective is persistently higher. `SelfTune` does not reduce the objective throughout the experiment.

Lastly, Fig. 4.11 shows how `pol/m3`'s parameter evolution over time. Most interpretable are the switch weights (Fig. 4.10a), which shift predictably to favor under-served classes in each regime (e.g., DSCP 10 in Phase II).



(a) No trust region (b) No correction

(c) No model (d) No denoiser

(e) No trust region (f) No correction

(g) No model (h) No denoiser

Figure 4.12: Per-class 99th percentile slowdowns (top) and switch weights (bottom) under ablation. Removing key components—like the trust region, performance model, or correction—degrades SLO compliance and stability. Note: Y-axis scales vary.

4.9.4 *Ablation study*

We use the same bursty workload and high tightness SLO thresholds as in §4.9.2. All experiments are based on `pol/m3`, but each omits a specific component:

- *No trust region (TR)* disables the trust region.

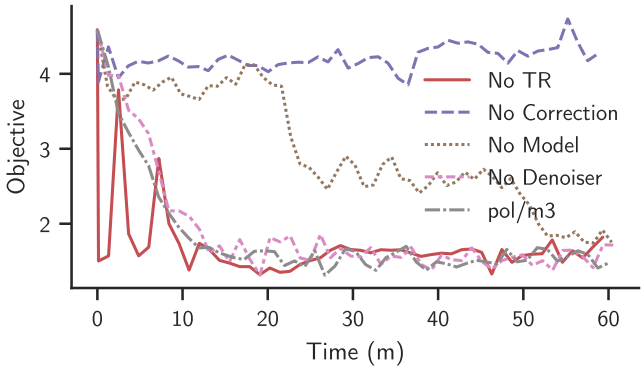


Figure 4.13: Global objective over time under component ablations. The full `Polyphony` controller (`pol/m3`) and no denoiser both converge; no trust region converges but with oscillation. No model converges more slowly, and no correction does not make progress.

- *No correction* disables the online model correction, relying solely on `m3`'s predictions.
- *No model* removes the model (`m3`), forcing the controller to explore directly on the real system.
- *No denoiser* feeds raw (unsmoothed) slowdown measurements to the controller.

Fig. 4.12 presents per-class slowdown trajectories and corresponding weight adjustments. Removing the trust region (Fig. 4.12a) results in larger parameter changes and early SLO violations (especially for DSCP 18 and 26), with weight oscillations between extremes. Without residual correction (Fig. 4.12b) the biased simulator mis-orders nearby points, and fails to improve the objective within a trust region. Without the model (Fig. 4.12c), the controller must explore many more configurations before it finds a good one, missing SLOs for the entire duration. The "no denoiser" variant performs well but exhibits high-frequency jitter in slowdowns and weights.

Fig. 4.13 summarizes the global objective under each variant. `pol/m3` converges smoothly. "No model" and "no correction" both plateau early, with the latter performing the worst overall. "No TR" reduces the objective, but with high initial variance.

4.9.5 *Scalability analysis in ns-3*

While our CloudLab [29] experiments validate `Polyphony` in realistic and noisy conditions, they are limited to a small topology. We conduct a scalability analysis using ns-3 [98].

We construct 24 simulation scenarios by varying the following parameters:

- *Topology*: We use a 32-rack, 256-host fat-tree topology sampled from Meta's data center networks [9].

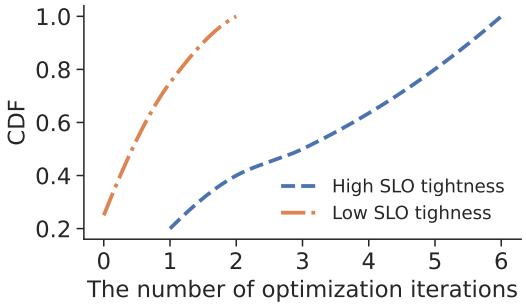


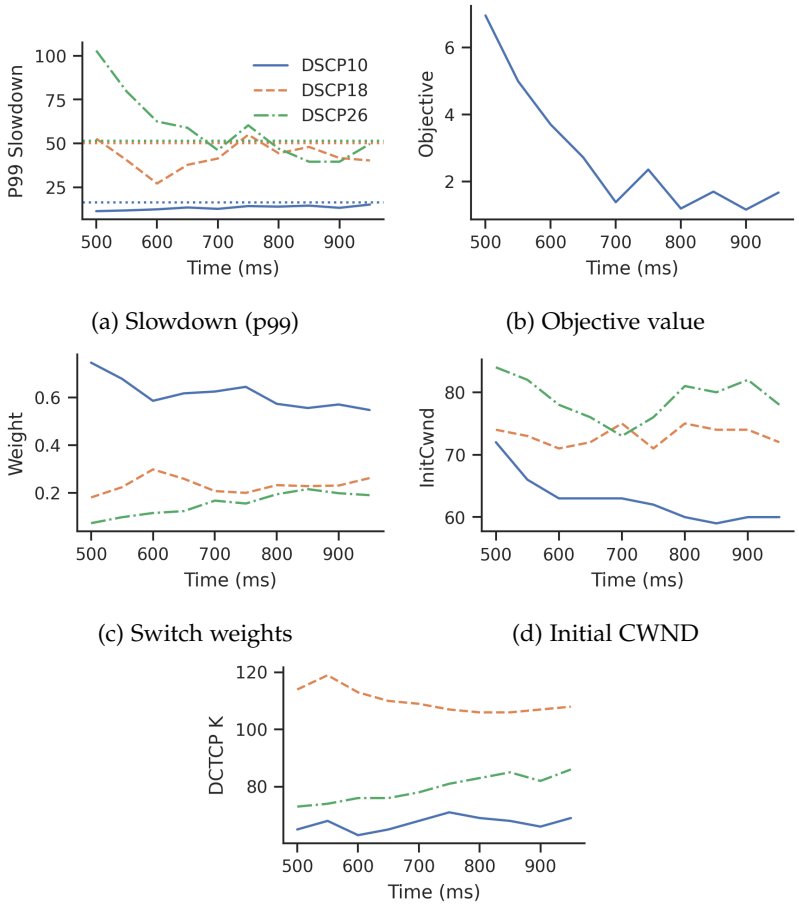
Figure 4.14: CDF of convergence time across ns-3 scenarios; scenarios that do not converge are omitted.

- *Workload:* We generate flow-level traffic using two flow size distributions derived from publicly available production workloads (`WebSearch` and `Hadoop` [74]). Inter-arrival times follow a log-normal distribution with $\sigma = 1$ (low burstiness) or $\sigma = 2$ (high burstiness) [95]. The network load is fixed at 60%. We use three application-specific traffic matrices, database, web server, and Hadoop clusters [74], to capture realistic rack-to-rack communication patterns. Flow endpoints are selected uniformly at random from hosts within each rack.

- *SLO tightness*: For each scenario, we randomly select an SLO tightness level by scaling a baseline slowdown by 25% (high tightness) or 50% (low). Baseline slowdowns are obtained by running a simulation under a fixed, near-optimal configuration. The 99th percentile slowdown from this simulation is used as the baseline SLO and scaled accordingly.

We evaluate `pol/m3` based on two metrics: *SLO compliance rate* and *convergence time* across all scenarios. Each simulation runs for 1 second. The first 500 ms are used to gather an initial set of datapoints. Starting at 500 ms, `pol/m3` selects a new configuration every 50 ms.

Fig. 4.15 visualizes a representative scenario: the controller reduces 99th percentile flow slowdowns (a) and global objective (b) while making coordinated updates to switch weights (c), initial CWNDs (d), and DCTCP thresholds (e). Fig. 4.14 shows the CDF of convergence times across the 19 scenarios where `pol/m3` successfully converged. `pol/m3` achieves 80% (19/24) SLO compliance. Despite variations in workload characteristics and SLO tightness, `pol/m3` stabilizes to SLOs in most cases: under low SLO tightness, almost all scenarios converge within two optimization iterations; under high SLO tightness, convergence takes longer but still completes within six iterations in all cases.



(a) Slowdown (p99)

(b) Objective value

(c) Switch weights

(d) Initial CWND

(e) DCTCP K

Figure 4.15: `pol/m3`'s behavior in ns-3 [98]. **(a)** Per-class 99th percentile flow completion time slowdowns; dotted lines indicate SLO thresholds. **(b)** Global objective improves steadily over time. **(c–e)** Controller updates switch weights, initial congestion windows, and DCTCP thresholds smoothly to meet SLOs.

4.10 RELATED WORK

4.10.1 *Tuners and controllers*

BLACKBOX AUTOTUNERS. A large body of work automates configuration tuning by treating the system as a black box and running controlled trials to search high-dimensional knob spaces. Representative systems include CherryPick [4] for cloud instance choices, Metis [59] and OtterTune [84] for service/DBMS parameters, OPPerTune [81] and SelfTune [52] for post-deployment service and cluster-manager knobs, AutoSched [37] for deep-learning schedulers, and FLASH [72] as a sample-efficient Bayesian optimization method. These approaches are supervisory and episodic: they propose a configuration, measure the live system, and iterate to convergence—often over minutes to hours—and can induce temporary regressions during exploration.

They are effective for finding good static settings but are not designed to regulate a performance metric on a tight feedback loop.

ONLINE CONTROLLERS. A separate line uses continuous feedback to control a system variable directly. For example, Autothrottle employs a bi-level design with a contextual-bandit making minute-by-minute CPU-throttle targets for microservices to satisfy latency SLOs [86], and ACC applies deep RL to adjust ECN thresholds at switches to keep queues short while sustaining throughput [90]. These are online regulators of specific mechanisms (CPU throttling; ECN marking) rather than broad configuration search. Polyphony sits in this controller category: it targets network QoS metrics directly (e.g., class-specific p99) and selects actions on a minute timescale, but, unlike prior online controllers, does so model-guided, using fast approximate performance models to predict counterfactual outcomes and reduce the need for risky live experimentation.

4.10.2 *Simulation-based optimization and control*

Simulation offers a scalable and low-risk means to explore system behavior, especially in complex networking environments. Modern network simulators use parallelization [11, 36, 95] and ML techniques [32, 56, 91, 93] to model packet-level behavior with improved speed and fidelity. Despite these advances, a gap remains between simulated and real-world performance due to unmodeled hardware interactions and environmental variability [33]. To bridge this gap, ByteDance's Crescent [38] emulates production switch software in isolated environments, enabling realistic and low-risk evaluation of network configurations. Using simulations and emulations as predictive models, Model Predictive Control (MPC) offers formal methods for proactive configuration planning. Zipper [14], for instance, applies MPC to dynamically allocate bandwidth in 5G RAN slicing, using model-based forecasts to maintain SLO compliance. However, conventional MPC approaches require sufficiently accurate predictions; robust variants exist, but building fast, globally accurate models in data center networks is an open problem. `Polyphony` addresses this problem with an online corrective overlay to compensate for modeling errors, and it further applies trust-region-based optimization strategies to focus exploration in regions where predictions are most reliable.

4.11 CONCLUSION

This section described the design, implementation, and evaluation of `Polyphony`, a system for controlling tail latency at minute-scale to achieve performance objectives for different traffic classes—even under dynamic workloads. `Polyphony` leverages fast but inaccurate models

to estimate the consequence of applying a proposed configuration change. To correct for model error, `Polyphony` builds and applies a corrective overlay anchored on the best known alternative; Bayesian optimization is then used within a region around that alternative. Provided that the prediction model is reasonably accurate, e.g., using machine learning, `Polyphony` can quickly converge to a more optimal operating point and adapt to changing conditions. Under tight SLOs, noisy measurements, and large workload shifts, `Polyphony` stabilizes to meet SLOs within fifteen minutes.

5 DISCUSSION

This dissertation proposed, designed, and evaluated *prediction-guided control*, an approach that enables data center networks to meet tail latency SLOs with less provisioned capacity, or to achieve tighter performance targets with the same capacity. It works by continuously measuring workloads and tail latency, and by continuously regulating network parameters—congestion control thresholds, initial congestion windows, and switch queue weights. The central challenge is predicting the effect of parameter changes: they interact in complex, nonlinear ways, making trial-and-error tuning both slow and risky. To address this, we introduced `Parsimon` (Chapter 3), a fast tail-latency simulator that runs orders of magnitude faster than full-fidelity models while preserving distribution-level accuracy. Yet even fast, accurate simulators can produce wrong answers when embedded in a control loop, as we saw in §4.2.1. `Polyphony` therefore wraps such approximate models in a closed-loop controller that corrects for residual error online, explores safely within a trust region, and resets when workload regimes shift. Together, these contributions trace a path from prediction to control: fast models enable many "what-if" trials, and online feedback makes them reliable enough to steer real systems toward SLOs. This chapter reflects on the limitations of this approach and discusses future work and broader implications.

5.1 SUMMARY OF LIMITATIONS

While prediction-guided control offers a promising path forward, it is important to understand when and why it may not work. In this section, we synthesize the limitations of `Parsimon` and `Polyphony`.

5.1.1 *`Parsimon`'s limitations*

`Parsimon`'s speed comes from an independence approximation in which it simulates each link in isolation and combines the results. Consider what happens in a real network when a flow traverses multiple hops. Congestion upstream can smooth the traffic that arrives downstream, spacing packets more evenly and reducing queueing at later hops. By simulating each link in isolation, `Parsimon` misses this effect and slightly overestimates delay. The reverse can also occur: in networks with Priority Flow Control (PFC), downstream congestion can pause upstream links entirely, propagating backpressure in ways that independent simulation cannot capture. Lastly, congestion else-

where in the network can have similar smoothing effects on flows that later intersect a given path.

Correlation presents another challenge. In networks where traffic intensities across hops are correlated, a flow that encounters congestion at one hop is likely to encounter it at the next; conversely, a flow that sees an empty queue at one hop will likely see empty queues downstream. In statistical terms, correlation pushes probability mass from the center of the delay distribution toward the tails. Independent simulation, by construction, does not capture this effect. Finally, aggregating per-link delays by convolution overcounts delay for long flows that occupy multiple queues at the same time. §3.3.6 discusses these error sources in detail, and §3.5.7 presents experiments showing how and when accuracy degrades.

`Parsimon` also assumes static single-path routing. It models ECMP, but the route for a particular flow is decided once and does not change. Techniques like packet spraying, flowlet switching, and adaptive routing make path decisions at runtime based on instantaneous queue depths or link loads and are thus incompatible with `Parsimon`'s routing assumptions. Networks that rely heavily on these mechanisms are still best served by a full-fidelity simulator like ns-3.

Beyond these fundamental limitations, `Parsimon` does not model end-host dynamics like operating system scheduling jitter, garbage collection pauses, and application-level delays, which can dominate application performance in some cases. These effects could be added with significant engineering effort (much as they could be added to ns-3), but doing so would likely slow simulation considerably, and the more fundamental limitations would still remain.

Finally, model speed and accuracy directly impact convergence when `Parsimon` is used within a control loop. In Chapter 4, `Parsimon` required approximately 95 seconds per control step, compared to 7 seconds for `m3`. Moreover, because `m3` was trained against the actual testbed, it exhibited significantly higher accuracy than `Parsimon` for that deployment. As a consequence, `Polyphony` using `Parsimon` converged only for relatively easy SLOs and required more time to reach compliance. In the workload adaptation experiments (§4.9.3), `Polyphony` with `Parsimon` outperformed a model-free reinforcement learning baseline but converged much more slowly than `Polyphony` with `m3`. This illustrates the *value of prediction*: faster, more accurate models enable faster, safer convergence. When models are slower or less accurate, operators must widen control windows, shrink step sizes, and accept slower SLO recovery.

5.1.2 *Polyphony's limitations*

Polyphony's effectiveness is tied to the quality of its underlying predictor, as discussed above. Beyond model quality, Polyphony faces several challenges that stem from its high-level design.

First, it lacks formal guarantees. Polyphony uses a trust region to ensure that each control step does not degrade performance in expectation, given what it currently believes about the system. This provides a safety net: the controller will not stray far from known-good configurations. However because Polyphony makes a sequence of local decisions, always choosing the best step forward from its current position, it can settle into a local optimum. There is no assurance that the final configuration is globally optimal, nor is there a bound on how long convergence will take. Convergence speed depends on many factors: the accuracy of the model, the speed of the model, the volatility of the workload, and how aggressively the trust region allows exploration. In stable regimes with good models, Polyphony converges quickly. In noisy or shifting conditions, it may take more steps. If the workload changes faster than Polyphony can adapt, it may never converge.

The second challenge is scalability. Polyphony models the residual error between predictions and observations using Gaussian processes. These are effective in moderate dimensions of up to 20 or 30 parameters, but struggle as the number of tunable knobs grows. This is a well-known limitation of methods based on Gaussian processes: the cost of inference scales poorly, and the sample complexity increases with dimensionality. For networks where operators wish to tune hundreds of parameters simultaneously, like per-application parameters or per-switch parameters in a large network, Polyphony would need either a different surrogate model or a way to decompose the problem into smaller subdomains that can be optimized independently.

Third, Polyphony depends on accurate workload characterization. Our prototype uses Workfeed, a telemetry system that samples flow events to create traffic distributions. We validated this approach in CloudLab testbed experiments, where it successfully detected workload shifts and allowed Polyphony to adapt (§4.9.3). But our testbed comprised only a small handful of hosts, not tens of thousands. While Workfeed's sampling and aggregation strategies are designed to scale to larger deployments, we did not have the infrastructure to validate it at true data center scale.

Moreover, to mirror the structure of our workload generator, we built Workfeed to characterize traffic at the TCP flow level, where each flow represents a separate connection. In production data centers, however, many RPCs often multiplex over persistent TCP connections. A single long-lived TCP flow may carry hundreds or thousands of RPCs with different sizes and timing characteristics. To apply this approach in

such environments, Workfeed would likely need to be extended to track RPC-level events rather than TCP flow-level events. This could require instrumentation at the application or RPC framework layer, or inference at the TCP layer based on the timing of adjacent RPCs. Additionally, Workfeed's sampling and Poisson-based re-expansion preserve basic workload statistics but do not capture fine-grained temporal structure, such as synchronized bursts or causally-related flows arriving close together in time (such as during incast). When these patterns significantly affect queueing behavior, the reconstructed traces may underestimate peak congestion even if aggregate statistics are correct.

Finally, Polyphony implicitly assumes that no other controller will intefere with its own operation. In production data centers, this may not hold. Traffic engineering systems periodically adjust routing weights to balance load. Auto-scaling services add or remove hosts in response to demand. Load balancers shift traffic between replicas. Each of these systems operates its own control loop, and their decisions can interact. Interference from other control loops makes the environment appear time-varying from Polyphony's perspective even if the underlying system is stable, and since Polyphony relies on sufficient stationarity to learn a corrective residual model, uncoordinated controllers can undermine its convergence. Coordinating multiple controllers—deciding who adjusts what, and when—is a broader problem that we do not address. For now, Polyphony works best when it has sole authority over the parameters it tunes, or when other control loops operate in sufficiently different domains that their interactions are negligible.

5.2 FUTURE WORK

The limitations above suggest several directions for future work.

VALIDATION AT SCALE. We validated Polyphony and Workfeed on a small CloudLab testbed comprising a handful of hosts. While its sampling and aggregation strategies are designed to scale to larger deployments, we did not have the infrastructure to validate it at true data center scale with tens of thousands of hosts. Testing Workfeed and Polyphony in production-scale environments would reveal whether the sampling rates, aggregation latencies, and controller response times remain practical under realistic traffic volumes and network sizes. It would also expose whether the overhead of flow monitoring becomes prohibitive, and whether the controller can maintain stable operation in the presence of more heterogeneous workloads and topologies. Validating on incast-heavy workloads would also be valuable, as these create severe tail-latency spikes; doing so would require extending Workfeed to capture more fine-grained temporal structure, as discussed below.

SCALING TO MORE PARAMETERS. `Polyphony` uses Gaussian processes to model residual error, which limits its scalability to moderate dimensions—typically 20 to 30 parameters. For networks where operators wish to tune hundreds of parameters simultaneously, `Polyphony` would require either a different surrogate model or a way to decompose the problem. Examples include supporting a larger number of traffic classes or adding per-class bandwidth limits to prevent high-priority long flows from starving other traffic. More broadly, nothing in the design of networks requires configurations to be applied uniformly: it may be appropriate to tune end-host parameters on a per-application basis and switch parameters on a per-switch basis. However, this significantly increases the number of tunable knobs. One promising direction is to exploit structure in the parameter space: many parameters may be conditionally independent or affect only local subsets of traffic. By decomposing the optimization into smaller, independent or loosely coupled subproblems, `Polyphony` could scale to larger parameter spaces without sacrificing convergence speed or sample efficiency.

DISCRETE PARAMETER TYPES. `Polyphony` currently handles continuous parameters. In practice, parameters like switch weights, ECN thresholds, and initial congestion windows are quantized, but we have found that continuous relaxation has worked well. Many network configuration choices, however, are truly categorical: selecting a specific congestion control protocol or queueing discipline, choosing between ECMP and weighted load balancing, or enabling or disabling a protocol feature. These have no natural ordering and cannot be meaningfully interpolated. A straightforward extension would be to one-hot encode each categorical choice and treat the resulting binary dimensions as additional continuous inputs to the surrogate model. More sophisticated approaches could model categorical and continuous parameters jointly, using mixed-variable optimization techniques or surrogate models designed for heterogeneous parameter types.

ENHANCED WORKLOAD TRACKING. To match the structure of our workload generator, `Workfeed` defines a flow to be a TCP connection, even though in RPC systems, a single connection will be multiplexed across many RPCs. Extending `Workfeed` to track RPC-level events rather than TCP flows would enable more accurate workload characterization in production environments. One approach is to instrument the application or RPC framework layer, such as hooks into gRPC, Thrift, or proprietary RPC systems. An alternative is to infer RPC boundaries from timing data at the TCP layer. Beyond RPC tracking, `Workfeed`'s sampling and re-expansion currently preserve basic workload statistics but do not capture fine-grained temporal structure. Because we replicate flows independently, patterns like synchronized

bursts or causally-related flows arriving together can be lost. When these patterns significantly affect queueing behavior, such as with incast patterns, the reconstructed traces may underestimate peak congestion. Future work could explore reconstruction methods that preserve these temporal correlations; this would likely require tracking additional metadata about flow relationships and arrival patterns.

COORDINATING WITH OTHER CONTROLLERS. `Polyphony` implicitly assumes it has sole authority over the parameters it tunes. In production data centers, however, multiple control loops operate concurrently: traffic engineering systems adjust routing weights, autoscaling services add or remove hosts, and load balancers shift traffic between replicas. If these systems operate on similar timescales and make conflicting adjustments, they may interfere with each other, causing oscillations or slower convergence. Future work could explore coordination mechanisms that allow multiple controllers to coexist safely. One approach is to partition the parameter space into disjoint sets, where each controller has exclusive authority over its own parameters: `Polyphony` tunes congestion control and queueing, traffic engineering adjusts routing weights, and auto-scaling manages capacity. However, partitioning parameters alone is not sufficient; each controller's objective must also primarily depend on its own parameters rather than those controlled by others. For example, if multiple controllers all aim to minimize tail latency but adjust different knobs, they may still interfere, as changes by one controller could undo the work of another. Partitioning both parameters and objectives—so that each controller optimizes a metric primarily sensitive to its own knobs—would allow independent operation without conflict. It may be necessary to arrange controllers hierarchically, with higher-level controllers setting objectives or constraints that lower-level controllers respect. Another interesting opportunity is joint controller design: for example, the prediction engine could help make better task placement or autoscaling decisions by being network-aware.

EXTENDING MODEL SCOPE. `Parsimon` is constrained by what it models: it assumes shortest-path routing and does not support multipathing, adaptive routing, or load-balancing schemes that dynamically split flows across paths. It also does not model end-host delays, like operating system scheduling, garbage collection, or NIC offload behaviors, which can dominate application performance in some deployments. Extending `Parsimon` to cover a broader range of interventions would require nontrivial implementation effort, and some interventions may fall outside what its per-link decomposition can model effectively. A machine-learned model like `m3` can more easily capture end-host delays, since it trains on real system measurements that implicitly include these effects, but it is less clear how to extend either

approach to handle multi-pathing and adaptive routing because both assume a single path per flow.

5.3 BROADER IMPLICATIONS

While not explored in this dissertation, prediction-guided control is not specific to tail latency in data center networks. More broadly, it applies whenever operators must steer a complex software system with many interacting parameters, limited headroom for trial-and-error, and nonstationary workloads. In such settings, deriving faithful first-principles models is impractical: unlike many physical systems with relatively stable governing equations, software systems evolve and resist durable analytic models. Instead, one can pair fast approximate predictors—built from simulation, microbenchmarks, or learned surrogates—with an online correction scheme that treats predictions as priors rather than oracles. In our case, this corresponds to using a `Parsimon`- or `m3`-like predictor together with `Polyphony`'s residual learning, trust regions, and resets. The controller uses the predictor to explore a rich configuration space safely and cheaply, then closes the loop by learning residual error, constraining moves within a trust region, and resetting or tracking when regimes shift due to workload changes, software updates, or deployment reconfigurations.

As examples, consider LLM serving systems, which must adjust batching strategies, cache management, and quantization to meet P99 latency SLOs under shifting query patterns. Operating system schedulers face a similar challenge: tuning time slices and affinity policies to balance tail latency and fairness across heterogeneous workloads. The same pattern plausibly extends to cluster autoscaling and load balancing, stream processing pipelines, storage engines, and datacenter power management, where parameters such as DVFS settings and server power states shape both efficiency and performance. Each of these domains resists analytical modeling but could permit fast approximate prediction and control.

This hypothesis has boundaries. The approach requires that the system be controllable, in the sense that there must be parameters whose adjustment meaningfully affects the metrics of interest, and observable, in the sense that metrics can be measured at the cadence of control decisions. It also requires that predictions complete faster than the control horizon, so the controller can evaluate candidate configurations before deploying them. The approach is less suitable when feedback is too slow or too noisy to distinguish good changes from bad ones, or when workloads shift so rapidly that patterns do not persist long enough for effective learning. Establishing generality beyond networks is promising future work.

6 CONCLUSION

Modern web services decompose requests into thousands of RPCs, and tail latencies dominate end-to-end performance. Operators set tail latency SLOs to meet user expectations, but enforcing these SLOs under shifting workloads usually requires provisioning for peak load. Data center networks offer a rich control surface—congestion control thresholds, initial congestion windows, switch queue weights—that could be tuned to meet SLOs without overprovisioning, but manual tuning is impractical at scale. Model-free autotuners avoid manual work but explore through trial-and-error, risking SLO violations and converging slowly. This dissertation presents prediction-guided control as an alternative: fast approximate models explore the configuration space cheaply, while online feedback corrects for prediction error and adapts to workload changes.

`Parsimon` (Chapter 3) decomposes a data center network into independent per-link simulations that run in parallel. This decomposition sacrifices per-flow precision for distribution-level accuracy, preserving tail latency estimates while running approximately 500 times faster than ns-3, enabling "what-if" exploration at speeds compatible with online control loops. However, this speed comes at the cost of accuracy: like most simulators, `Parsimon` does not model many real-world effects, leading to higher error when compared to measurements from real testbeds.

`Polyphony` (Chapter 4) wraps approximate models like `Parsimon` and `m3`—a subsequent machine-learned predictor trained directly on real hardware—in a closed-loop controller that treats predictions as priors rather than oracles. It uses Gaussian processes to learn the residual error between model predictions and live measurements, optimizes within a trust region to ensure safe exploration, and detects workload shifts to reset when the operating regime changes. Because `m3` is trained on the target deployment, it is both faster and more accurate than `Parsimon`: in experiments using `m3` as the underlying predictor, `Polyphony` converged to tight SLOs within minutes when workloads shifted, whereas SelfTune, a model-free reinforcement learning baseline, struggled to adapt. When using `Parsimon`, `Polyphony` converged more slowly due to the model's lower accuracy and longer iteration time, but still outperformed the model-free baseline. These results demonstrate that faster, more accurate models enable faster convergence, but also that closed-loop correction can make even coarse predictors useful for control.

Many challenges remain. Extending this approach to production scale will require validating workload collections on tens of thousands of hosts and tracking RPC-level events rather than TCP flows. Scaling the controller to handle more parameters would enable finer-grained tuning, such as per-application or per-switch settings. Coordinating `Polyphony` with other control loops—like those found in traffic engineering, autoscaling, and load balancing—requires developing an architecture for multi-controller coordination to avoid interference. Beyond data center networks, prediction-guided control may apply wherever operators must tune complex software systems with non-stationary workloads and limited tolerance for trial-and-error, such as LLM serving, operating system scheduling, or data center power management, though establishing this generality remains future work.

Together, `Parsimon` and `Polyphony` trace a path from prediction to control: fast models enable cheap exploration, and online feedback corrects modeling errors, continuously steering the network to meet tail latency SLOs.

BIBLIOGRAPHY

[1] John Aitchison. "The statistical analysis of compositional data." In: *Journal of the Royal Statistical Society: Series B (Methodological)* 44.2 (1982), pp. 139–160.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. "Hedera: Dynamic flow scheduling for data center networks." In: *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*. Vol. 10. 8. San Jose, USA. 2010, pp. 89–92.

[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 59–76.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics." In: *Proceedings of the USENIX NSDI*. 2017, pp. 469–482.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. "CONGA: Distributed congestion-aware load balancing for datacenters." In: *Proceedings of the ACM SIGCOMM 2014 Conference*. 2014, pp. 503–514.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data center TCP (DCTCP)." In: *Proceedings of the ACM SIGCOMM 2010 Conference*. 2010, pp. 63–74.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. "Less is more: Trading a little bandwidth for ultra-low latency in the data center." In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 253–266.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. "pFabric: Minimal near-optimal datacenter transport." In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 435–446.

[9] Alexey 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/. 2014.

[10] Microsoft Azure. *Service Level Agreement for Azure Cosmos DB*. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/. 2024. (Visited on 05/03/2025).

[11] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. "Unison: A parallel-efficient and user-transparent network simulation kernel." In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 115–131.

[12] Wei Bai et al. "Empowering Azure Storage with RDMA." In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 49–67. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/bai.

[13] Hari Balakrishnan, Venkata N Padmanabhan, and Randy H Katz. "The effects of asymmetry on TCP performance." In: *Mobile Networks and Applications* 4.3 (1999), pp. 219–241.

[14] Arjun Balasingam, Manikanta Kotaru, and Paramvir Bahl. "Application-level service assurance with 5G RAN slicing." In: *Proceedings of the USENIX NSDI 24*. 2024, pp. 841–857.

[15] Forest Baskett, K Mani Chandy, Richard R Muntz, and Fernando G Palacios. "Open, closed, and mixed networks of queues with different classes of customers." In: *Journal of the ACM (JACM)* 22.2 (1975), pp. 248–260.

[16] David L. Black, Zheng Wang, Mark A. Carlson, Walter Weiss, Elwyn B. Davies, and Steven L. Blake. *An architecture for differentiated services*. RFC 2475. Dec. 1998. DOI: 10.17487/RFC2475. URL: https://www.rfc-editor.org/info/rfc2475.

[17] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[18] Robert T. Braden, Dr. David D. Clark, and Scott Shenker. *Integrated services in the Internet architecture: An overview*. RFC 1633. June 1994. DOI: 10.17487/RFC1633. URL: https://www.rfc-editor.org/info/rfc1633.

[19] Robert T. Braden, Lixia Zhang, Steven Berson, Shai Herzog, and Sugih Jamin. *Resource reservation protocol (RSVP) – Version 1 functional specification*. RFC 2205. Sept. 1997. DOI: 10.17487/RFC2205. URL: https://www.rfc-editor.org/info/rfc2205.

[20] Eric Brandwine. *A day in the life of a billion requests: AWS authentication system evolution*. Talk, AWS re:Invent 2022, session SEC404. Slides/video state the IAM pipeline handles "over half a billion requests per second." 2022. URL: https://www.youtube.com/watch?v=tPr1AgGkvc4 (visited on 05/03/2025).

[21] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning." In: *CoRR* abs/1012.2599 (2010). arXiv: 1012.2599. URL: http://arxiv.org/abs/1012.2599.

[22] Jake Brutlag. *Speed matters for Google web search*. Tech. rep. Google Inc., 2009. URL: https://services.google.com/fh/files/blogs/google_delayexp.pdf (visited on 05/03/2025).

[23] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. "Overload control for μs-scale RPCs with Breakwater." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 299–314.

[24] *Congestion control in IP/TCP internetworks*. RFC 896. Jan. 1984. DOI: 10.17487/RFC0896. URL: https://www.rfc-editor.org/info/rfc896.

[25] Rene L. Cruz. "A Calculus for Network Delay, Part I: Network Elements in Isolation." In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 114–131.

[26] Rene L. Cruz. "A Calculus for Network Delay, Part II: Network Analysis." In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 132–141.

[27] Jeffrey Dean and Luiz André Barroso. "The tail at scale." In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[28] Alan Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and simulation of a fair queueing algorithm." In: *ACM SIGCOMM Computer Communication Review* 19.4 (1989), pp. 1–12.

[29] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. "The design and operation of CloudLab." In: *Proceedings of the USENIX ATC*. 2019, pp. 1–14.

[30] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. "Scalable global optimization via local Bayesian optimization." In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/6c990b7aca7bc7058f5e98ea909e924b-Paper.pdf.

[31] Rob Ewaschuk and Betsy Beyer. "Monitoring Distributed Systems." In: *Site Reliability Engineering: How Google Runs Production Systems*. Ed. by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. Chapter 6. O'Reilly Media, 2016. URL: https://research.google/pubs/monitoring-distributed-systems/.

[32] Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Krzysztof Rusek, Shihan Xiao, Xiang Shi, Xiangle Cheng, Pere Barlet-Ros, and Albert Cabellos-Aparicio. "RouteNet-Fermi: Network modeling with graph neural networks." In: *IEEE/ACM Transactions on Networking* 31.6 (2023), pp. 3080–3095.

[33] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated design of self-adaptive software with control-theoretical formal guarantees." In: *Proceedings of the International Conference on Software Engineering*. 2014, pp. 299–310.

[34] Daniel 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.

[35] Sally Floyd and Van Jacobson. "Link-sharing and resource management models for packet networks." In: *IEEE/ACM transactions on Networking* 3.4 (1995), pp. 365–386.

[36] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. "DONS: Fast and affordable discrete event network simulation with automatic parallelization." In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023, pp. 167–181.

[37] Wei Gao, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, and Tianwei Zhang. "Autosched: An adaptive self-configured framework for scheduling deep learning training workloads." In: *Proceedings of the ACM International Conference on Supercomputing*. 2024, pp. 473–484.

[38] Zhaoyu Gao, Anubhavnidhi Abhashkumar, Zhen Sun, Weirong Jiang, and Yi Wang. "Crescent: emulating heterogeneous production network at scale." In: *Proceedings of the USENIX NSDI*. 2024, pp. 1045–1062.

[39] Google Cloud Platform Blog. *gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments*. 2016. URL: https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments (visited on 05/03/2025).

[40] Google Cloud. *Use the Cloud Monitoring dashboard*. Last updated 2025-11-24, accessed 2025-11-26. 2025. URL: https://docs.cloud.google.com/datastore/docs/use-monitoring-dashboard.

[41] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. "Backpressure flow control." In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 779–805.

[42] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 1–14.

[43] Dr. Roch Guerin, Dr. Craig Partridge, and Scott Shenker. *Specification of guaranteed quality of service*. RFC 2212. Sept. 1997. DOI: 10.17487/RFC2212. URL: https://www.rfc-editor.org/info/rfc2212.

[44] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant." In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.

[45] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin 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. Association for Computing Machinery, 2017, pp. 29–42.

[46] Félix Hernández-Campos. *Generation and validation of empirically-derived TCP application workloads*. The University of North Carolina at Chapel Hill, 2006.

[47] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. "Finishing flows quickly with preemptive scheduling." In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 127–138.

[48] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN." In: *Proceedings of the ACM SIGCOMM 2013 Conference*. 2013, pp. 15–26.

[49] Christian Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. Nov. 2000. DOI: 10.17487/RFC2992. URL: https://www.rfc-editor.org/info/rfc2992.

[50] James R Jackson. "Networks of waiting lines." In: *Operations Research* 5.4 (1957), pp. 518–521.

[51] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. "B4: Experience with a globally-deployed software defined WAN." In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–14.

[52] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. "SelfTune: Tuning cluster managers." In: *Proceedings of the USENIX NSDI*. 2023, pp. 1097–1114.

[53] Frank P Kelly. "Networks of queues." In: *Advances in Applied Probability* 8.2 (1976), pp. 416–432.

[54] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. "Swift: Delay is simple and effective for congestion control in the datacenter." In: *Proceedings of the ACM SIGCOMM 2020 Conference*. 2020, pp. 514–528.

[55] Rémi Lafage. "egobox, a Rust toolbox for efficient global optimization." In: *Journal of Open Source Software* 7.78 (2022), p. 4737. DOI: 10.21105/joss.04737. URL: https://doi.org/10.21105/joss.04737.

[56] Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. "m3: Accurate flow-level performance estimation using machine learning." In: *Proceedings of the ACM SIGCOMM*. 2024, pp. 813–827.

[57] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. "Tales of the tail: Hardware, OS, and application-level sources of tail latency." In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. 2014, pp. 1–14.

[58] Yuliang Li et al. "HPCC: High precision congestion control." In: *Proceedings of the ACM SIGCOMM 2019 Conference*. 2019, 44–58.

[59] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. "Metis: Robustly tuning tail latencies of cloud systems." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 981–992.

[60] Chung Laung Liu and James W Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[61] Michael Marty et al. "Snap: A microkernel approach to host networking." In: *In ACM SIGOPS 27th Symposium on Operating Systems Principles*. New York, NY, USA, 2019.

[62] Jim McManus, Joseph Malcolm, Michael D. O'Dell, Daniel O. Awduche, and Johnson Agogbua. *Requirements for Traffic Engineering Over MPLS*. RFC 2702. Sept. 1999. DOI: 10.17487/RFC2702. URL: https://www.rfc-editor.org/info/rfc2702.

[63] Vishal Misra, Wei-Bo Gong, and Don Towsley. "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED." In: *Proceedings of the ACM SIGCOMM 2000 Conference*. 2000, pp. 151–160.

[64] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. "TIMELY: RTT-based congestion control for the datacenter." In: *Proceedings of the ACM SIGCOMM 2015 Conference*. 2015, 537–550.

[65] Jeffrey C Mogul and John Wilkes. "Nines are not enough: Meaningful metrics for clouds." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 136–141.

[66] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A receiver-driven low-latency transport protocol using network priorities." In: *Proceedings of the ACM SIGCOMM 2018 Conference*. 2018, pp. 221–235.

[67] John Nagle. "On packet switches with infinite storage." In: *IEEE transactions on communications* 35.4 (1987), pp. 435–438.

[68] Abhay K. Parekh and Robert G. Gallager. "A generalized processor sharing approach to flow control in integrated services networks: the single-node case." In: *IEEE/ACM Transactions on Networking* 1.3 (1993), pp. 344–357.

[69] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. "Fastpass: A centralized "zero-queue" datacenter network." In: *Proceedings of the ACM SIGCOMM 2014 Conference*. 2014, pp. 307–318.

[70] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. "Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking." In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 66–85.

[71] Prometheus Authors. *Prometheus: Monitoring system and time series database*. https://prometheus.io. Accessed: 2025-04-24. 2024.

[72] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. "FLASH: Fast model adaptation in ML-centric cloud platforms." In: *Proceedings of Machine Learning and Systems* 6 (2024), pp. 524–544.

[73] Dr. K. K. Ramakrishnan and Sally Floyd. *A proposal to add explicit congestion notification (ECN) to IP*. RFC 2481. Jan. 1999. DOI: 10.17487/RFC2481. URL: https://www.rfc-editor.org/info/rfc2481.

[74] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. "Inside the Social Network's (Datacenter) Network." In: *Proceedings of the ACM SIGCOMM 2015 Conference*. 2015, pp. 123–137.

[75] Harshit Saokar, Margot Leibold, et al. "ServiceRouter: Hyperscale and minimal-cost service mesh at Meta." In: *Proc. USENIX OSDI*. 2023. URL: https://www.usenix.org/system/files/osdi23-saokar.pdf (visited on 05/03/2025).

[76] Linus Schrage. "A proof of the optimality of the shortest remaining processing time discipline." In: *Operations Research* 16.3 (1968), pp. 687–690.

[77] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. "A cloud-scale characterization of remote procedure calls." In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 498–514. ISBN: 9798400702297. DOI: 10.1145/3600006.3613156. URL: https://doi.org/10.1145/3600006.3613156.

[78] Madhavapeddi Shreedhar and George Varghese. "Efficient fair queueing using deficit round robin." In: *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 1995, pp. 231–242.

[79] Arjun Singh et al. "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network." In: *Proceedings of the ACM SIGCOMM 2015 Conference*. 2015, 183–197.

[80] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. "Programmable packet scheduling at line rate." In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 44–57.

[81] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. "OPPerTune: Post-deployment configuration tuning of services made easy." In: *Proceedings of the USENIX NSDI*. 2024, pp. 1101–1120.

[82] Ion Stoica, Scott Shenker, and Hui Zhang. "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks." In: *Proceedings of the ACM SIGCOMM 1998 Conference*. 1998, pp. 118–130.

[83] Amin Vahdat. *Speed, scale and reliability: 25 years of Google datacenter networking evolution*. 2024. URL: https://cloud.google.com/blog/products/networking/speed-scale-reliability-25-years-of-data-center-networking (visited on 05/03/2025).

[84] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. "Automatic database management system tuning through large-scale machine learning." In: *Proceedings of the ACM international conference on management of data*. 2017, pp. 1009–1024.

[85] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. "More is less: Reducing latency via redundancy." In: *Proc. ACM HotNets*. 2012. URL: https://conferences.sigcomm.org/hotnets/2012/papers/hotnets12-final34.pdf (visited on 05/03/2025).

[86] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. "Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices." In: *Proceedings of the USENIX NSDI*. 2024, pp. 149–165.

[87] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. "Better never than late: Meeting deadlines in datacenter networks." In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 50–61.

[88] Robert Winter et al. *Ethernet Jumbo Frames*. http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf. 2009.

[89] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C Lee. "Rajomon: Decentralized and coordinated overload control for latency-sensitive microservices." In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 2025, pp. 21–36.

[90] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. "ACC: Automatic ECN tuning for high-speed datacenter networks." In: *Proceedings of the ACM SIGCOMM*. 2021, pp. 384–397.

[91] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. "DeepQueueNet: Towards scalable and generalized network performance estimation with packet-level visibility." In: *Proceedings of the ACM SIGCOMM*. 2022, pp. 441–457.

[92] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. "High-resolution measurement of data center microbursts." In: *Proceedings of the 2017 Internet Measurement Conference*. Nov. 2017, pp. 78–85. DOI: 10.1145/3131365.3131375.

[93] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. "MimicNet: Fast performance estimates for data center networks with machine learning." In: *Proceedings of the ACM SIGCOMM 2021 Conference*. 2021, pp. 287–304.

[94] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. "Aequitas: Admission control for performance-critical RPCs in datacenters." In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 1–18.

[95] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. "Scalable tail latency estimation for data center networks." In: *Proceedings of the USENIX NSDI*. 2023, pp. 685–702.

[96] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. "Overload control for scaling WeChat microservices." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 149–161.

[97] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. "Congestion control for large-scale RDMA deployments." In: *Proceedings of the ACM SIGCOMM 2015 Conference*. 2015, 523–536.

[98] *ns-3 Network Simulator*. https://www.nsnam.org. 2020.