Application-Aware Scheduling Architectures for Mobile Systems

by

Arjun Balasingam

B.S., Stanford University (2018)
S.M., Massachusetts Institute of Technology (2021)

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Arjun Balasingam. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Arjun Balasingam
Department of Electrical Engineering and Computer Science
January 26, 2024

Certified by: Hari Balakrishnan
Fujitsu Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Studies

Application-Aware Scheduling Architectures for Mobile Systems

by

Arjun Balasingam

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Abstract

Architects of mobile systems have long optimized schedulers for platform-level objectives, such as system throughput or operating cost. However, these objectives could be at odds with performance indicators that applications or users of the platform might care about. This thesis proposes two-tiered architectures to realize app-aware resource allocation policies for mobile systems. The first level decomposes app-level objectives into platform-level objectives over scheduling rounds. The second level leverages classical schedulers, designed for platform objectives, as building blocks to guide the optimizer toward app-level objectives. We apply this design paradigm to build resource allocation systems and algorithms in two domains: mobility platforms and cellular networks.

Mobius allocates tasks from different customers to vehicles in mobility platforms, which are used for food and package delivery, ridesharing, and mobile sensing. Over rounds, Mobius invokes vehicle routing solvers that maximize task completion throughput to compute schedules that are fair to different customers using the platform. On a trace of Lyft rides in New York City, Mobius computes max-min fair online schedules involving 200 vehicles and over 16,000 tasks, while achieving only 10% less throughput than a classical vehicle routing solver.

Zipper is a radio resource scheduler that fulfills throughput and latency service-level agreements for individual apps connected to a cellular network. Zipper bundles apps into network slices, and leverages classical schedulers that maximize base station throughput to compute resource schedules for each slice that comply with each app's requirements. On a typical workload consisting of video streaming, conferencing, IoT, and virtual reality apps, Zipper reduces tail throughput and latency violations, measured as a ratio of violation of the app's request, by 9×, compared to traditional base station schedulers.

Thesis supervisor: Hari Balakrishnan
Title: Fujitsu Professor of Electrical Engineering and Computer Science

To my parents, Esha and Pratheep.

Acknowledgments

Graduate school has been the most intellectually-rewarding few years of my life, and I owe much of the credit to my advisor, Hari Balakrishnan. He created the space for me to explore an exceptionally broad range of topics over the last five and a half years. Besides the two projects discussed in this dissertation, we worked on drone computing and sensing, on improving bicycle safety with LiDAR, and on keypoint tracking in videos. I have had a blast working with him, and resonate deeply with his philosophy of applications motivating research problems. Hari has taught me how to communicate complex research ideas clearly and succinctly. Moreover, he has been an incredibly supportive mentor to me, both professionally and personally.

I am fortunate to have also worked closely with the other members of my thesis committee. Mohammad Alizadeh has been a phenomenal co-advisor. I have learned so much from him about how to be a better researcher. He has a knack for explaining complex and messy systems with simplicity and elegance. Mohammad has also been extremely patient with me, and given thoughtful and constructive feedback in every one of our meetings. I am privileged to have had Victor Bahl also mentor me through much of my graduate studies, beginning with my first internship at Microsoft in the summer of 2020. Despite having a busy day job, Victor has been incredibly generous with his time to me. I am grateful to him for creating an opportunity for me to see firsthand how to translate cutting-edge industry research to product. Radhika Mittal was my very first collaborator when I came to MIT. We worked together on drone computing, which at the time was a research area that was new to both of us. She helped me conceive and formalize the idea of fairness in mobility

platforms, which eventually formed the basis for Mobius.

I consider myself lucky to have had a great support system throughout my career as a student researcher. I am grateful to Saman Amarasinghe, Frans Kaashoek, and Devavrat Shah[1] for their advice over the years. My undergraduate mentors, Sachin Katti and Aaron Schulman, encouraged me to pursue a doctorate, and introduced me to the mobile networking community, which has been my research home since 2016. They trusted me to lead my own project as an undergrad, and I believe that this early experience prepared me well for graduate school. Many thanks, also, to Manu Bansal, Sam Joseph, and Rakesh Misra for their friendship and advice ever since we first met at Stanford 8 years ago.

I have had many fantastic collaborators throughout graduate school: Mohammad Alizadeh, Venkat Arun, Paramvir Bahl, Hamsa Balakrishnan, Hari Balakrishnan, Favyen Bastani, Michael Cafarella, Joseph Chandler, Karthik Gopalakrishnan, Songtao He, Ziwen Jiang, Manikanta Kotaru, Tim Kraska, Chenning Li, Sam Madden, Radhika Mittal, Ahmed Saeed, and Zhoutong Zhang. I am especially grateful to Karthik and to Mani, who persevered with me through the thick of the Mobius and Zipper projects, respectively. Zhoutong generously brought me up to speed on keypoint tracking techniques being developed by computer vision researchers. I am also fortunate to have had the opportunity to mentor some junior students during my time at MIT; notably, the MicroTel and DriveTrack projects would not have been possible without Joe's tireless efforts debugging PyTorch and JAX training code.

Thanks to everyone in the Networks and Mobile Systems group and to my officemates in 32-G982, past and present: Venkat Arun, Frank Cangialosi, Inho Cho, Manya Ghobadi, Prateesh Goyal, Pouya Hamadanian, Songtao He, Pantea Karimi, Mehrdad Khani, Moein Khazraee, Sunghyun Kim, Chenning Li, Charlie Liu, James Lynch, Hongzi Mao, Radhika Mittal, Akshay Narayan, Arash Nasr-Esfahany, Vikram Nathan, Parimarjan Negi, Kimia Noorbakhsh, Amy Ousterhout, Seo Jin Park, Sudarsanan Rajasekaran, Ahmed Saeed, Harsha Sharma, Vibhaa Sivaraman, Will Sussman, Shaileshh Venkatakrishnan, Frank Wang,

[1] Thanks to Frans and Devavrat for many thrilling squash matches!

Lei Yang, Mingran Yang, and Zhizhen Zhong. I will look back fondly on our group events, including hikes, rock climbing sessions, impromptu ping-pong games, and virtual game nights.

My first winter in Boston was miserable. But the city has slowly grown on me over the last few years, and I will deeply miss living here. Thanks to my friends for a memorable time in Boston: Yamin Arefeen, Jialin Ding, Siddhartha Jayanti, Akshay Narayan, Sarath Pattathil, Aniruddh Raghu, Nalini Singh, Vibhaa Sivaraman, Abhin Shah, Sohil Shah, Kush Tiwary, and Kapil Vaidya. I will cherish the many squash matches, badminton games, soccer scrimmages, (bitterly cold!) winter morning runs, fall foliage hikes, and Tosci's & New City ice cream trips that we have enjoyed over these last five and a half years.

I owe an immense debt of gratitude to my family. My sister, Ramya Balasingam, and brother, Akhilesh Balasingam, have made my frequent trips home worth it, always finding ways to keep my mind off of research. I spent 18 months of the pandemic back home in California, and my two siblings made that extended stay at home enjoyable. My parents, Esha and Pratheep Balasingam, have been my two biggest cheerleaders. Since I was young, they encouraged me to be fearless and curious, and pushed me to dream big. They have always made education a priority for me and my siblings. I cannot thank them enough for all of their sacrifices and for their unconditional support over the years. I dedicate this thesis to them.

Previously Published Material

Chapter 3 revises a paper published at ACM MobiSys 2021 [10]:

> Arjun Balasingam, Karthik Gopalakrishnan, Radhika Mittal, Venkat Arun, Ahmed Saeed, Mohammad Alizadeh, Hamsa Balakrishnan, and Hari Balakrishnan. Throughput-fairness tradeoffs in mobility platforms. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 363–375, New York, NY, USA, 2021. Association for Computing Machinery

Chapter 4 revises a paper that will appear at USENIX NSDI 2024 [12]:

> Arjun Balasingam, Manikanta Kotaru, and Paramvir Bahl. Application-level service assurance with 5G RAN slicing. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, USA, April 16-18, 2024*. USENIX Association, 2024

Contents

Acknowledgments **7**

List of Figures **17**

List of Tables **23**

1 Introduction **25**

1.1 Strawman: End-to-End Optimization 27

1.2 Application-Aware Scheduling Architectures 27

1.2.1 Customer-Level Fairness in Mobility Platforms 29

1.2.2 Application-Level Service Assurance in Cellular Networks 30

1.3 Organization of this Thesis 31

2 Background and Related Work **33**

2.1 Hierarchical Scheduling 33

2.2 Scheduling in Mobility Platforms 34

2.3 Service Assurance in Cellular Networks 35

3 Customer-Level Fairness in Mobility Platforms **39**

3.1 Problem Setup 43

3.2 Overview 45

3.3 Balancing Throughput and Fairness 47

3.3.1 Scheduling on the Convex Boundary 48
3.3.2 Scheduling in Dynamic Environments 50
3.3.3 Visualizing Routes Scheduled by Mobius 51
3.4 Mobius Scheduling Algorithm 51
3.4.1 Finding Support Allocations 52
3.4.2 Scheduling Over Rounds 55
3.4.3 Optimality of Mobius 56
3.4.4 Implementation 58
3.5 Generalizing to α-Fairness 59
3.6 Real-World Evaluation 61
3.6.1 Online Trace-Driven Emulation 61
3.6.2 Case Study: Lyft Ridesharing in Manhattan 63
3.6.3 Case Study: Shared Aerial Sensing Platform 69
3.7 Conclusion 74

4 Application-Level Service Assurance with 5G RAN Slicing 77

4.1 Problem Setup and Challenges 80
4.1.1 Problem Formulation 81
4.1.2 Challenge: State Space Complexity 83
4.1.3 Challenge: Determining RAN Resource Availability 84
4.2 Design 86
4.2.1 Model Predictive Control 86
4.2.2 Tuning Slice Bandwidths Efficiently 89
4.2.3 Forecasting RAN Resource Availability 92
4.3 Implementation 95
4.4 Evaluation 97
4.4.1 Evaluation Setup 98
4.4.2 End-to-end Evaluation 99

4.4.3 SLA Compliance 100
4.4.4 Forecasting RAN Resource Availability 106
4.4.5 Microbenchmarks 109
4.5 Discussion 110
4.6 Conclusion 111

5 Conclusion **113**
5.1 Summary 113
5.2 Future Work 114

A Mobius Appendix **117**
A.1 Searching for α-Fair Allocation 118
A.2 Mobius Algorithm 119
A.3 Optimality of Mobius 120
A.3.1 Mobius is Optimal in a Round 120
A.3.2 Mobius Converges to the Target Throughput 122
A.4 Greedy Heuristic to Maximize U_α 125
A.5 Runtime of Mobius 127
A.6 Microbenchmarks 128
A.6.1 Robustness to Spatial Demand 128
A.6.2 Expressive Schedules with α 130
A.6.3 Timescale of Fairness 131
A.6.4 Geometry of the Convex Boundary 132
A.6.5 Varying the Number of Vehicles 133
A.6.6 A Case with Three Customers 134

B Zipper Appendix **137**
B.1 Allocating Slice Bandwidth in Zipper 137
B.1.1 Forecasting the Wireless Channel with an RNN 137

B.1.2 Monotonicity of Throughput and Latency 138
B.1.3 Algorithm 138
B.2 Estimating Resource Availability in Zipper 138
B.2.1 DNN Architecture 138

References **141**

List of Figures

1-1 This thesis proposes a two-leveled architecture for app-aware scheduling problems in mobile systems, where the higher-layer algorithm guides a blackbox lower-layer platform scheduler to fulfill app-level objectives. 28

3-1 An example with two customers, two vehicles, and a 6-minute planning horizon. Mobius computes a schedule that (i) achieves a similar total throughput to that of the max throughput schedule, and (ii) preserves the customer-level fairness achieved by the round-robin and dedicated schedules. 43

3-2 Imposing fairness at short timescales (e.g., one round trip) degrades throughput. Executing Options 1 and 2 provides fairness at longer timescales and leads to greater total throughput. 45

3-3 In each round, Mobius uses a VRP solver to compute a schedule that maximizes a weighted sum of throughputs, and automatically adjusts the weights across rounds to improve fairness. 46

3-4 Visualizing feasible allocations of throughput for a small problem with two customers and two vehicles. Allocations on the convex boundary trade short-term fairness for throughput. The convex boundary becomes denser over time, making the target allocation achievable. 47

3-5 The difference in spatial density of tasks leads to short-term unfairness (Rounds 1 and 3). Mobius compensates for this by directing more resources to the underserved customer (Round 2). 51
3-6 Using a blackbox VRP solver as a building block, Mobius runs an iterative search algorithm to find the support allocations. 54
3-7 Mobius (a) finds the support allocations nearest the target allocation in each round, and (b) converges to the target allocation. 56
3-8 Mobius can tune its allocation to deliver proportional fairness ($\alpha = 1$) and max-min fairness (approximated with $\alpha = 100$). 60
3-9 Maps of zones (customers) and demand in Manhattan, indicating skews in both spatial coverage and volume of ride requests. 64
3-10 Long-term throughputs for zones in Manhattan after 13 hours. A good scheduler should have a stacked plot with large evenly-sized blocks, and a map with bright (high throughput) and homogeneous (fair) colors across zones. 66
3-11 Time series of long-term throughputs for two zones for different replanning horizons. Frequent replanning ensures fairness (equal throughputs) at shorter timescales. 67
3-12 Distributions of rider wait times for two zones. Even though Mobius compromises some throughput for fairness, it delivers similar wait times as the max throughput scheduler. 68
3-13 Summary of aerial sensing applications, which span a variety of spatial demand and reactive/continuous sensing preferences. We collected ground truth data for each of these applications using real drones, and created traces to evaluate Mobius. 68
3-14 Map of tasks for 5 aerial sensing apps, spanning a 1 square mile area in Cambridge, MA. Mobius replans every 5 minutes, in order to incorporate new requests. Each drone returns to recharge every 15 minutes. 70

3-15 Long-term throughputs achieved over 90 minutes. Mobius achieves high throughput and best shares it amongst the apps. 72
3-16 Percentage of tasks completed per app. Mobius fulfills nearly all requests for the Traffic and Parking apps, before allocating "excess" vehicle time to the more backlogged apps. 73
3-17 Discounting long-term throughput allows Mobius to gradually respond to the sudden presence of the transient Roof app, instead of dedicating all drones to it. 74

4-1 Apps express their connectivity requirements in terms of SLAs, and the operator provisions slice bandwidths to fulfill all SLAs. 79
4-2 Zipper can efficiently manage an expressive and comprehensive state space to deliver SLAs for each app in each slice. 81
4-3 Translating an app's SLAs directly to required slice bandwidth can ignore schedules with greater spectral efficiency. 84
4-4 Zipper provisions connectivity by dynamically optimizing network slice bandwidth and resource allocation to meet app-level SLAs. 86
4-5 Zipper uses model predictive control (MPC) to compute slice bandwidths that comply with all app SLAs. With MPC, Zipper decouples prediction from control to manage the state space. 86
4-6 Zipper is resilient to modest $\sim$2 dB error in forecasting SNR. Its MPC framework supports different channel forecasters. While both have small median errors, the RNN model outperforms EWMA. 87
4-7 Exposing more bandwidth to a slice reduces packet latency. 90
4-8 Zipper builds a family of DNNs that forecasts bandwidth distributions for slices consisting of different MAC schedulers and apps with different demand patterns. 92
4-9 We implement Zipper atop a production-class 5G network. 95

4-10 Zipper tunes the bandwidth allocated to a slice serving a mobile OnePlus phone running 17 Mbps iPerf flow. 100
4-11 Tail throughput penalties for varying load. Apps scheduled by Zipper experience 95th percentile penalties close to 0%. 102
4-12 Tail latency penalties for varying base station loads. Apps scheduled by Zipper have low 95th and 99th percentile penalties. 103
4-13 Throughput for 75 apps + 20 best effort apps. Zipper meets the SLAs reliably, and allocates excess capacity to best effort. 104
4-14 Zipper's performance is invariant to the number of slices. 106
4-15 Both the conditional and DNN-based resource estimation methods achieve bounded (and low) penalties. 107
4-16 Zipper's DNN resource estimator achieves a higher admit rate and utilization by squeezing in apps with lighter demand. 108
4-17 Runtime of Zipper and NVS. Even though Zipper involves more computation than NVS, it is still practical for large workloads. 109

A-1 Proof setup for Lemma 7. The face $\overline{BE}$ is the same as in Fig. 3-7 124
A-2 Comparing customer throughputs and platform throughput achieved by Mobius and other schemes. Customer tasks stream in according to a static task arrival model. Mobius consistently outperforms other schemes by striking a balance between throughput and fairness. 128
A-3 Snapshot of per-round schedules computed by Mobius (for 3 rounds) and other policies. Mobius compensates for short-term unfairness by switching between schedules on the convex hull over rounds. Other schemes suffer from persistent bias or low throughput. 129

A-4 Mobius converges to the fair allocation of throughputs regardless of the timescale of fairness. Scheduling in shorter rounds converges faster to the fair allocation of rates, but longer round durations lead to schedules with greater platform throughput. 131
A-5 Convex boundaries computed by Mobius for the different maps shown in Fig. A-2a. The shape of the convex boundary describes the inherent tradeoff between fairness and high throughput. 132
A-6 Long-term per-customer rates computed by Mobius on Map D in Fig. A-2a, for different provisioning of vehicles. 133
A-7 Mobius vs. dedicating vehicles for example with 3 customers. Mobius converges to a fair allocation of rates for customers, when the assumption on static task arrival is relaxed. 134

B-1 Architecture of RNN model to forecast wireless channel. 137

List of Tables

1-1 This thesis applies the insight of hierarchical scheduling to build resource allocation systems and algorithms in two application domains. 29

4-9 Apps, SLAs, and frequencies selected for experiments. 98

A-1 Performance of Mobius on different input sizes. 127

Chapter 1

Introduction

Mobile systems form the fabric for the hyper-connected society in which we live today. We interface with our smartphones for routine services, such as remote meetings, online shopping, and telehealth checkups. Moreover, many physical devices, including security cameras, sensors, virtual reality headsets, and automobiles now connect to the Internet. Operators can remotely schedule vehicle fleets and robots powering food delivery, ridesharing, and mobile sensing applications. Over the last decade, mobile systems have become indispensable to our daily lives, even supporting mission-critical services, such as emergency response, mobile health, and secure financial transactions.

A vital component of any mobile system is its resource allocation policy, which typically answers the following question: *how should the system share its resources among its users?* Resources are usually scarce, forcing the operator to make important tradeoffs. For example, a mobile network operator like Verizon must allocate wireless bandwidth—the scarce resource—to different users subscribing to its network. Similarly, Uber Eats must allocate restaurant delivery tasks to vehicles, with fuel being the scarce resource. In each of these systems, the operator must decide how to allocate the scarce resource among its users in a given time period, while fulfilling the platform's operating goals.

An operator of a mobile system selects what resource allocation policy to deploy. Tradi-

tionally, these policies have optimized platform-level objectives. For instance, a base station operator would try to maximize total throughput or profit [74, 115], or minimize total cost or energy consumed. The operator of a mobility platform might fulfill tasks that require the least travel time or use up the least vehicle fuel [4, 54, 113]. These objectives directly address capital and operational expenditures, which matter most to system operators.

However, the users or applications of these mobile systems may not care about platform-level objectives—these metrics have little bearing on a given user's performance. For instance, a cloud gaming app may want low latency, a FaceTime call may want low jitter, a virtual reality (VR) application may want high bandwidth and low latency, and an internet-of-things (IoT) sensor needs low power consumption. Similarly, a traffic analysis app running on a shared drone computing platform may need fresh and timely video measurements of road segments to maintain up-to-date speed estimates. A restaurant like McDonald's may want a certain number of orders fulfilled by Uber Eats, and would not be satisfied if the platform instead decided to prioritize deliveries from Starbucks.

The challenge is that *app-level objectives are often at odds with platform-level objectives*. For instance, a base station operator might find that serving a bandwidth-hungry file transfer more resources than a lightweight IoT download keeps the link more occupied and leads to more profit. Uber Eats might conclude that fulfilling more Starbucks orders at the expense of McDonald's orders will allow it complete more deliveries for the same amount of vehicle fuel and earn more revenue. Fulfill app-level objectives requires trading off platform efficiency.

The performance and reliability of individual apps has only grown in importance, as we have begun relying on mobile systems for mission-critical services, especially during and after the COVID-19 pandemic. For instance, delivering food and medical supplies can often be time sensitive. Cellular networks, similarly, are critical infrastructure for telemedicine and autononmous vehicles, where unpredictable throughput and latency can have drastic consequences.

Moreover, mobile systems, in particular, suffer from variable and dynamic environments. For instance, base stations schedule resources in the presence of dynamic wireless channel

conditions, which affect the throughput and latency that end users might experience. Mobility platforms must account for traffic conditions, refueling constraints, and weather, when computing schedules for vehicle fleets. Computing resource schedules under these kinds of variability becomes more challenging when the objectives capture more fine-grained app-level performance metrics, as opposed to aggregate platform-level metrics.

1.1 Strawman: End-to-End Optimization

A natural approach to realize app-level objectives when allocating resources in mobile systems is to develop new end-to-end heuristics that allocate system resources to directly fulfill the requirements at hand. However, tailoring heuristics for specific objectives is tedious. Moreover, new heuristics lose the performance guarantees that come with well-studied platform-level scheduling algorithms.

Recently, reinforcement learning (RL) has grown in popularity for combinatorial optimization problems with new or different objective functions. Given a definition of the state space and the objective function expressed in terms of state variables, an RL agent can learn a suitable policy that best maximizes the objective function, by interacting with a simulated environment. The RL framework can adapt to new objective functions, making it attractive for scheduling problems with a variety of potential app-level objectives. However, training agents for these objectives is computationally expensive and the resulting policies generalize poorly, especially without a model of the environment, which is often hard to obtain.

1.2 Application-Aware Scheduling Architectures

This thesis introduces a two-layer scheduling paradigm to realize app-level resource allocation policies, illustrated in Fig. 1-1. The higher-layer algorithm determines the set of inputs into a platform-level scheduler that approximately maps to the desired app-level metrics. Across rounds, this algorithm refines these inputs to guide platform scheduler to realize

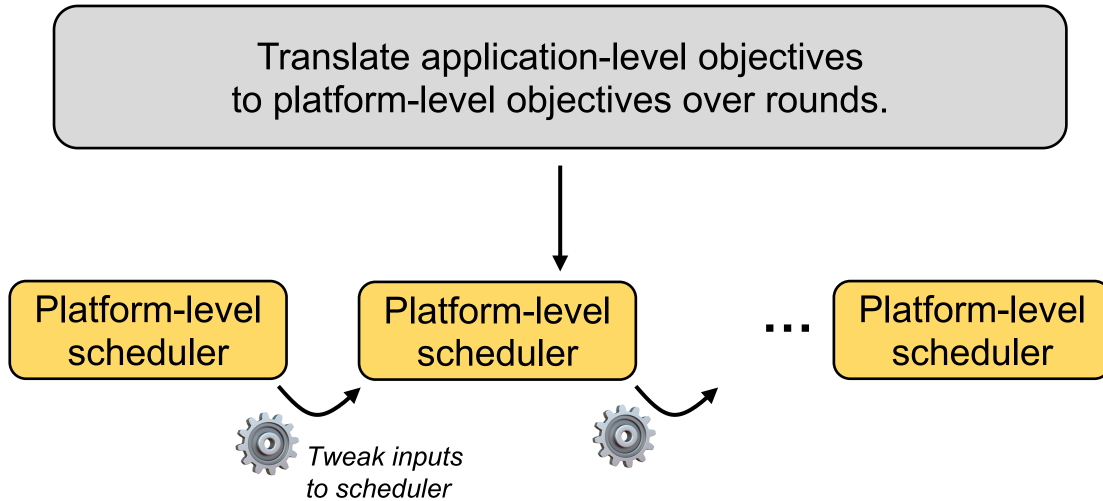


Figure 1-1: This thesis proposes a two-leveled architecture for app-aware scheduling problems in mobile systems, where the higher-layer algorithm guides a blackbox lower-layer platform scheduler to fulfill app-level objectives.

the app-level metrics. Our insight is that a higher-layer algorithm that guides a lower-layer unmodified platform scheduler strikes the right balance between app-level objectives and the overall efficiency of the system.

This modular architecture has multiple benefits. First, it narrows the scope of the resource allocation problem from an end-to-end behemoth to a narrow interface between app-level and platform-level objectives. Second, it allows operators to repurpose platform schedulers without modifying them. With this approach, operators can borrow the performance guarantees that come with well-studied platform schedulers, and do not to design new heuristics that capture the scheduling intricacies that are unique to a domain.

We leverage this insight to design app-aware scheduling systems and algorithms for two distinct domains: customer fairness in mobility platforms and app-level service assurance in cellular networks. Table 1-1 summarizes how each system fits into this paradigm. Between these two case studies, we also show that this hierarchical design structure generalizes to two different flavors of optimization: one that uses a single solver to realize all app-level objectives, and another that uses several instances of a solver to realize all app-level objectives.

	Mobius (Chapter 3)	**Zipper (Chapter 4)**
Domain	Mobility platforms	5G wireless
Objective	Customer fairness	Throughput/latency SLAs
Platform scheduler	Vehicle routing solver	Resource block scheduler
Search algorithm	Find customer-fair vehicle schedule by iteratively invoking solver for different combinations of task weights	Bundle apps into bandwidth slices and find SLA-compliant schedule and bandwidth allocation for each slice by iteratively invoking solver for different input bandwidths
Structure	Single global solver	One solver per slice

Table 1-1: This thesis applies the insight of hierarchical scheduling to build resource allocation systems and algorithms in two application domains.

1.2.1 Customer-Level Fairness in Mobility Platforms

Mobility platforms power applications like food and package delivery, ridesharing, and mobile sensing. We study the problem of allocating tasks from different customers (e.g., restaurants in a food delivery service or sensing apps in a drones-as-a-service deployment) to vehicles in these mobility platform. Ideally, a mobility platform should allocate tasks to vehicles and schedule them in order to optimize both throughput and fairness across customers. However, existing approaches to scheduling in this domain ignore fairness.

We introduce Mobius, a new scheduling system for mobility platforms built using a hierarchical design paradigm. Mobius uses guided optimization to achieve both high throughput and fairness across customers. Mobius divides the scheduling horizon into rounds, and produces the feasible set of allocations for that round using a standard throughput-maximizing vehicle routing solver [113]. Then, by dynamically tweaking weights into the optimizer, Mobius guides it toward a solution that is not feasible in the set for one round but can be achieved over multiple rounds.

Mobius supports spatiotemporally diverse and dynamic customer demands. It provides a principled method to navigate inherent tradeoffs between fairness and throughput caused by shared mobility. Our evaluation demonstrates these properties, along with the versatility and

scalability of Mobius, using traces gathered from ridesharing and aerial sensing applications. Our ridesharing case study shows that Mobius can schedule more than 16,000 tasks across 40 customers and 200 vehicles in an online manner.

1.2.2 Application-Level Service Assurance in Cellular Networks

5G promises to serve a wide variety of apps, such as mixed reality, cloud gaming, video conferencing, and cloud robotics, all of which require predictable network connectivity (e.g., throughput and latency). Ideally, a network operator should be able configure a network's resource allocation policy to cater to the specific connectivity requirements of each subscribing application. Typical base station schedulers optimize for coarse metrics, such as for the aggregate throughput at the base station or for the aggregate throughput achieved by a bundle of applications. However, none of these methods ensure adequate performance for *each* application connected to the network.

We introduce Zipper, a novel scheduling system for Radio Access Networks (RANs) that provides assurances of application-level throughput and latency. Zipper bundles apps into "network slices", a standards-compliant abstraction. Its high-layer scheduling algorithm dynamically assigns enough bandwidth to each slice so that each app meets its performance requirements. Each slice then uses standard throughput-maximizing resource block schedulers [74, 115] to allocate resources to apps in a slice. To select the bandwidths, Zipper casts the problem as a model predictive controller, explicitly modeling the network dynamics of each user. It then uses an efficient algorithm to compute slice bandwidth allocations that meet each app's requirements, invoking standard base station schedulers through an iterative search. To assist operators with interfacing admission control policies, Zipper exposes a primitive that estimates if there is bandwidth available to accommodate an incoming app's requirements.

We implemented Zipper on a production-class 5G virtual RAN testbed integrated with hooks to control slice bandwidth, and evaluated it on real workloads, including video conferencing and virtual reality apps. On a representative RAN workload, our real-time

implementation supports up to 200 apps and over 70 slices on a 100 MHz channel. Relative to a slice-level service assurance system, Zipper reduces tail throughput and latency violations, measured as a ratio of violation of the app's request, by 9×.

1.3 Organization of this Thesis

Chapter 2 provides background on hierarchical scheduling and discusses related work on resource allocation in mobility platforms and in cellular networks. Chapter 3 describes Mobius, a scheduler that provides customer-level fairness in mobility platforms. Chapter 4 presents Zipper, a 5G RAN slicing system that provides application-level service assurance. We build both systems with the unified insight that we can design app-aware resource allocation algorithms by decomposing the problem into smaller queries of a platform-level scheduler. Chapter 5 summarizes the contributions for this thesis and outlines directions for future work.

Chapter 2

Background and Related Work

2.1 Hierarchical Scheduling

Hierarchical scheduling is a popular strategy to prioritize tasks in different resource allocation settings, such as in real-time operating systems [119], multilevel queue scheduling [93], deadline-based scheduling [109], thread prioritization [71], and job scheduling in batch processing pipelines [128]. The two-layer structure, with a higher-layer global scheduler and a lower-layer task scheduler, is similar to the one proposed in this thesis. However, the global scheduler determines which app to serve and for how long [82], and the task scheduler decides how to allocate the system's resources among the app's tasks for the assigned time window.

However, the optimization objective in many of these prior methods relate to timing constraints. Both levels of the hierarchy have focused on decomposing time at different granularities, with the higher layer deciding time per app and the lower layer deciding time per task. In this thesis, we consider scheduling problems where the lower-layer platform scheduler optimizes a different objective from the higher-layer scheduler. For instance, Mobius (Chapter 3) guides a platform scheduler that maximizes task throughput to compute customer-fair schedules. Similarly, Zipper (Chapter 4) guides a platform scheduler that maximizes slice throughput to compute app SLA-compliant schedules.

Moreover, while this thesis also proposes running unmodified task scheduler at the lower layer, the interface between the higher-layer scheduler and the lower-layer (platform) scheduler is different. To realize app-level objectives, the systems in this thesis conduct an iterative search over different input parameters to the platform scheduler to guide the optimizer toward app-level metrics. The contribution of this thesis is a hierarchical scheduling paradigm that leverages lower-level schedulers built for platform objectives to realize different app-centric objectives.

2.2 Scheduling in Mobility Platforms

Shared mobility and sensing platforms

Ridesharing platforms rely on different flavors of the VRP; these systems have typically been interested in maximizing profit (i.e., throughput) [4, 22], minimizing the size of the fleet [118], and planning in an online fashion [16]. Similarly, there has been a large amount of recent work on drones-as-a-service platforms, which have primarily addressed challenges surrounding data acquisition [117], multi-tenancy and security [60], and programming interfaces [59, 90]. All of these systems use a throughput-maximizing algorithm under the hood. Mobius is motivated by the advent of *customer-centric* mobility platforms in a variety of domains, where guarantees on QoS to customers are paramount to the viability [116] of these services [88].

Vehicle routing problem

The VRP has been extensively studied by the Operations Research community [113]. Many variants of the problem have been considered, ranging from the budget-constrained VRP [8], capacitated VRP [54], VRP with time windows [44], predictive routing under stochastic demands [17, 59], etc. Prior work has extended the VRP to consider multiple objectives, such as minimizing the variance in vehicle travel time or tasks completed by each vehicle [70]. These load balancing objectives, however, do not consider customer-level fairness, which is

the focus of Mobius. Moreover, Mobius abstracts out fairness from the underlying vehicle scheduling problem, making its techniques complementary to the large body of work on the VRP and its variants.

Fair resource allocation in computer systems

Our approach to formalizing throughput and fairness in mobile task fulfillment is inspired by α-fair bandwidth allocation in computer networks [74, 92]. However, as noted earlier in this chapter, mobility platforms introduce new challenges around attributing cost to serve customers, that do not arise when addressing fairness in switch scheduling [38], congestion control [73], and multi-resource compute environments [53]. Mobius develops a novel set of techniques to address these challenges.

2.3 Service Assurance in Cellular Networks

RAN slicing

While a static allocation of PRBs to slices provides traffic isolation and simplifies radio resource management [106], it does not guarantee reliable slice performance, since throughput and latency vary with dynamic wireless channel conditions. Orion [48] and SCOPE [18] are slicing-capable RAN virtualization frameworks, and implement existing slice bandwidth schedulers like Network Virtualization Substrate (NVS) [76]. NVS, designed originally for WiMAX, allocates PRBs among slices to deliver a target aggregate slice throughput, assuming invariant Modulation and Coding Scheme (MCS) conditions. However, RAN operators adjust MCS according to time-varying channel conditions. Slice-level service assurance is also the primary focus of many other RAN slicing proposals [7, 29, 34, 80, 103].

RadioSaber [30], a recent RAN slicing system, allocates PRBs to slices in a channel-aware manner, by extending NVS to query each slice's Medium Access Control (MAC) scheduler and find the PRBs that are best-suited for each user's channels. RadioSaber is comple-

mentary to Zipper: it focuses on slice-level throughput assurance for enterprise slices, while Zipper is designed for app-level SLAs. Future work includes incorporating RadioSaber's techniques for channel-aware PRB allocation into Zipper.

LACO [130] proposes a reinforcement learning-based framework to provide latency guarantees in multi-tenancy environments by minimizing the number of bits missing the specified latency tolerance. By contrast, this paper presents a novel framework that tailors radio resource schedules for applications SLAs such as throughput and latency in the presence of dynamic wireless channel conditions.

Prior work [97] recognizes that slice-level assurance does not guarantee consistent performance for all users due to varying channel qualities within a slice, and evaluates under the context of a toy setting of airplane WiFi. Zipper's innovation lies in its app-level service assurance approach and end-to-end RAN slicing system. Zipper system goes beyond offline simulations, and (a) evaluates realistic 5G workloads on a production-class system with commercial UE and emulation, (b) enables customizable slice schedulers, and (c) demonstrates how to interface with admission controllers.

Admission control

Admission control proposals for RAN slicing cover traffic prediction [107], load balancing [24], pricing [96, 114], and game-theoretic formulations [25]. Zipper does not propose a new method for admission control; instead, we recognize that, to use Zipper on production networks, operators need to know if Zipper's slice controller has resources to accommodate the SLAs of an incoming app. Typical approaches to assess resource availability [56, 75] are not compatible with Zipper's app-level formulation; we elaborate in §4.1.3. Recent 5G network slicing proposals [30, 48, 130] do not address how to estimate resource availability, which is vital for operators to use these systems in practice.

RAN virtualization

Virtual RANs serve multiple logical RANs from the same physical hardware. They have garnered significant attention [35, 40, 121], with a number of proposals across different compute platforms, including CPUs [40, 51, 112, 123], DSPs [13] and GPUs [94]. Zipper leverages this body of work to dynamically allocate RAN resources among different virtual resources and expose them to the network slices.

Scheduling

Efficient RAN utilization is a key principle of mobile network design [57, 62]. Canonical algorithms, such as proportionally-fair [115], round-robin, and priority-based [129] schedulers, only consider aggregate throughput and fairness.

Chapter 3

Customer-Level Fairness in Mobility Platforms

The past decade has seen the rapid proliferation of mobility platforms that use a fleet of mobile vehicles to provide different services. Popular examples include package delivery (UPS, DHL, FedEx, Amazon), food delivery (DoorDash, Grubhub, Uber Eats), and rideshare services (Uber, Lyft). In addition, new types of mobility platforms are emerging, such as drones-as-a-service platforms [47, 60, 84, 124] for deploying different sensing applications on a fleet of drones.

In these mobility platforms, the vehicle fleet of cars, vans, bikes, or drones is a *shared infrastructure*. The platform serves multiple *customers*, with each customer requiring a *set of tasks* to be completed. For instance, each restaurant subscribing to DoorDash is a customer, with several food delivery orders (or tasks) in a city. Similarly, an atmospheric chemist and a traffic analyst might subscribe to a drones-as-a-service platform, each with their own sensing applications to collect air quality measurements and traffic videos, respectively, at several locations in the same urban area. Multiplexing tasks from different customers on the same vehicles can increase the efficiency of mobility platforms because vehicles can amortize their travel time by completing co-located tasks (belonging to either the same or different

customers) in the same trip.

We study the problem of scheduling spatially distributed tasks from multiple customers on a shared fleet of vehicles. This problem involves (i) assigning tasks to vehicles and (ii) determining the order in which each vehicle must complete its assigned tasks. The constraints are that each vehicle has bounded resources (fuel or battery). While several variants of this scheduling problem have been studied, the objective has typically been to complete as many tasks as possible in bounded time, or to maximize aggregate throughput (task completion rate) [54, 113].

We identify a second—equally important—scheduling requirement, which has emerged in today's customer-centric mobility platforms: *fairness* of customer throughput to ensure that tasks from different customers are fulfilled at similar rates.[1] For example, in food delivery, the platform should serve restaurants equitably, even if it means spending time or resources on restaurants with patrons far from the current location of the vehicles. A ridesharing platform should ensure that riders from different neighborhoods are served equitably, which ridesharing platforms today do not handle well, a phenomenon known as "destination discrimination" [88, 116, 131].

We seek an online scheduler for mobility platforms that achieves both high throughput and fairness. A standard approach to achieving these goals is to track the resource usage and work done on behalf of different users in a fine-grained way and equalize resource consumption across users. Such fine-grained accounting and attribution is difficult with shared mobility: the resource used is a moving vehicle traveling toward its next task, but making that trip has a knock-on benefit, not only for the next task served, but for subsequent ones as well. However, the benefit of a specific trip is not equal across the subsequent tasks. Although it may be possible to develop a fair scheduler that achieves high throughput using fine-grained accounting and attribution, it is likely to be complex.

We turn, instead, to an approach that has been used in both societal and computing

[1] The method we develop also applies to weighted fairness.

systems: optimization, which may be viewed as a search through a set of feasible schedules to maximize a utility function. In our case, we can establish such a function, optimize it using both the task assignment and path selection, and then route vehicles accordingly.

In a typical mobility problem, the planning time frame for optimization could be between 30 minutes and several hours, involving hundreds of vehicles, dozens of customers, and tens of thousands of tasks. The scale of this problem pushes the limits of state-of-the-art vehicle routing solvers [16]. Moreover, fairness objectives lead to nonlinear utility functions, which make the optimization much more challenging. As a benchmark, optimizing the routes for 3 vehicles and 17 tasks over 1 hour, using the CPLEX solver [65] with a nonlinear objective function, takes over 10 hours [89].

To address these problems, a natural approach is to divide the desired time duration into shorter rounds, and then run the utility optimization. When we do this, something interesting emerges in mobility settings: the space of feasible solutions—each solution being an achievable set of rates for the customers—often *collapses into a rather small and disturbingly suboptimal set!* These feasible solutions are either fair but with dismal throughput, or with excellent throughput but starving several customers.

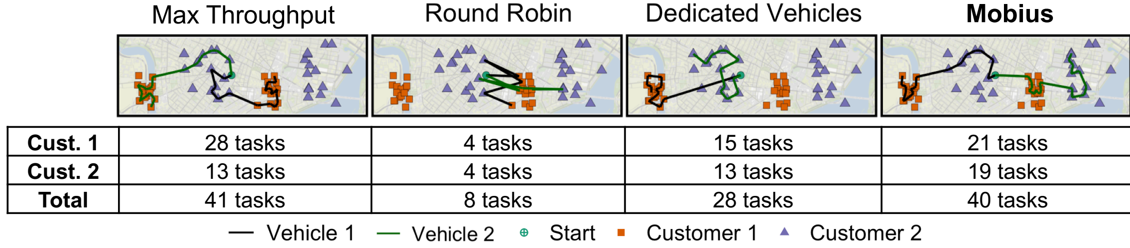
A simple example helps see why this happens. Consider a map with three areas, A_1, A_2, A_3, each distant from the others. There are several tasks in each area: in A_1, all the tasks are for customer C_1; in A_2, all the tasks are for customer C_2, and in A_3, all the tasks are for two other customers, C_3 and C_4. Suppose that there are two vehicles. Over a duration of a few minutes, we could either have the two vehicles focus on only two areas, achieving high throughput but ignoring the third area and reducing fairness, or, we could have them move between areas after each task to ensure fairness, but waste a lot of time traveling, degrading throughput. It is not possible here to achieve both throughput and fairness *over a short timescale*. Yet, over a long time duration, we can swap vehicles between regions to amortize the movement costs. This shows that planning over a longer timescale permits feasible schedules that are better than what a shorter timescale would permit.

Our contribution, *Mobius*, divides the desired time duration into rounds, and produces the feasible set of allocations for that round using a standard optimizer. Mobius guides the optimizer toward a solution that is not in the feasible set for one round but can be achieved over multiple rounds. This guiding is done by aiming for an objective that maximizes a weighted linear sum of customer rates in each round. The weights are adjusted dynamically based on the long-term rates achieved for each customer thus far. The result is a practical system that achieves high throughput and fairness over multiple rounds. This approach of achieving long-term fairness by setting appropriate weights across rounds allows us to use off-the-shelf solvers for the weighted Vehicle Routing Problem (VRP) for path planning in each round. Importantly, this design allows Mobius to optimize for fairness in the context of any VRP formulation, making this work complementary to the vast body of prior work on vehicle routing algorithms [4, 8, 17, 54].

Scheduling over multiple rounds also allows Mobius to handle tasks that arrive dynamically or expire before being done. Moreover, Mobius supports a tunable level of fairness modeled by α-fair utility functions [74], which generalize the familiar notions of max-min and proportional fairness.

We have implemented Mobius and evaluated it via extensive trace-driven emulation experiments in two real-world settings: (i) a ridesharing service, based on real Lyft ride request data gathered over a day, ensuring fair QoS to different neighborhoods in Manhattan; and (ii) urban sensing using drones for measuring traffic congestion, parking lot occupancy, cellular throughput, and air quality. We find that:

1. Relative to a scheduler that maximizes only throughput, Mobius compromises only 10% of platform throughput in order to enforce max-min fairness.
2. Compared to dedicating vehicles to customers, Mobius improves vehicle utilization by 30-50% by intelligently sharing vehicles amongst customers.
3. Mobius can compute fair online schedules at a city scale, involving 40 customers, 200 vehicles, and over 16,000 tasks.



	Max Throughput	Round Robin	Dedicated Vehicles	Mobius
Cust. 1	28 tasks	4 tasks	15 tasks	21 tasks
Cust. 2	13 tasks	4 tasks	13 tasks	19 tasks
Total	41 tasks	8 tasks	28 tasks	40 tasks

Figure 3-1: An example with two customers, two vehicles, and a 6-minute planning horizon. Mobius computes a schedule that (i) achieves a similar total throughput to that of the max throughput schedule, and (ii) preserves the customer-level fairness achieved by the round-robin and dedicated schedules.

3.1 Problem Setup

Every customer subscribing to a mobility platform submits several requests over time. Each request specifies a task (e.g., gather sensor data or deliver package) and a corresponding location. The platform schedules trips for each vehicle over multiple rounds. It takes into account any changes in a customer's requirements (in the form of new task requests or expiration of older unfulfilled tasks) at the beginning of each round. We say that a customer has a backlog if they have more tasks than can be completed by all available resources within the allocated time. For simplicity of exposition, we assume each customer is backlogged (our evaluation in §3.6 relaxes this assumption).

Let K be the set of customers, and $T_k(\tau)$ be the set of tasks requested by customer k during a scheduling round τ. We denote $x_k(\tau)$ as the throughput achieved for customer k in scheduling round τ, i.e., the total number of tasks in $T_k(\tau)$ that are fulfilled divided by the round duration.

We denote $\overline{x}_k(t)$ as the long-term throughput for each customer k, after t scheduling rounds, i.e., $\overline{x}_k(t) = \frac{1}{t}\sum_{\tau=1}^{t} x_k(\tau)$ if rounds are of equal duration. A good scheduling algorithm should achieve the following objectives:

- **Platform Throughput.** Maximize the total long-term throughput after round t, i.e., $\sum_{k \in K} \overline{x}_k(t)$.

- **Customer Fairness.** For any two customers $k_1, k_2 \in K$ with backlogged tasks, ensure $\overline{x}_{k_1}(t) = \overline{x}_{k_2}(t)$.

Equalizing long-term per-customer throughputs $\overline{x}_k(t)$ provides a desirable measure of fairness for many mobility platforms: higher per-customer throughputs correlate with other performance metrics, such as lower task latency and higher revenue. Our evaluation (§3.6) quantifies the impact of optimizing for a fair allocation of throughputs on other platform-specific quality-of-service metrics.

Prior algorithms for scheduling tasks on a shared fleet of vehicles have focused on the VRP, i.e., only considered maximizing platform throughput [54, 113]. Achieving per-customer fairness introduces three new challenges:

Challenge #1: Attributing vehicle time to customers

Vehicle time and capacity are scarce. Consider the example in Fig. 3-1, with two customers and two vehicles; customer 1 has two densely-packed clusters of tasks, while customer 2 has two dispersed clusters of tasks. We show schedules and tasks fulfilled by Mobius and three other policies: (i) maximizing throughput, (ii) dedicating a vehicle per customer, and (iii) alternating round-robin between customer tasks. Notice that, to the left of the depot (center of the map), customer 2's tasks can be picked up on the way to customer 1's tasks. Thus, multiplexing both customers' tasks on the same vehicle is more desirable than dedicating a vehicle per customer, because it amortizes resources to serve both customers. However, sharing vehicles amongst customers complicates our ability to reason about fairness, because the travel time between the tasks of different customers cannot be attributed easily to each one.

Challenge #2: Timescale of fairness

Fig. 3-2 shows two customers and one vehicle that must return home to refuel. A high-throughput schedule would dedicate the vehicle to one of the customers. By contrast, a fair schedule would require the vehicle to round-robin customer tasks, achieving low throughput

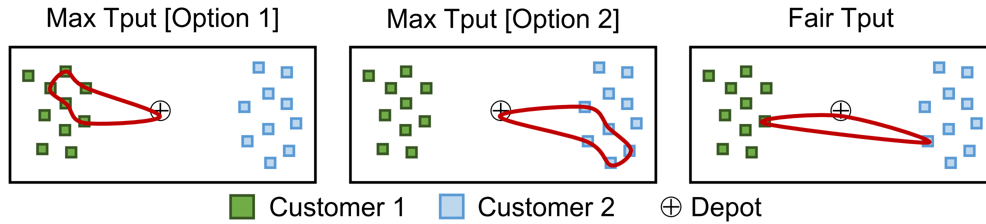


Figure 3-2: Imposing fairness at short timescales (e.g., one round trip) degrades throughput. Executing Options 1 and 2 provides fairness at longer timescales and leads to greater total throughput.

due to travel. Over a longer time duration, however, we can execute two max-throughput schedules (Options 1 and 2) to achieve both fairness and high throughput.

Challenge #3: Spatiotemporal diversity of tasks

In Fig. 3-1, the two customers' tasks have different spatial densities. The high-throughput schedule favors customer 1. A max-min fair schedule should, by contrast, ensure that customer 2 gets its fair share of the throughput, even if it comes at the cost of higher travel time and lower platform throughput. Striking the right balance between fairly serving a customer with more dispersed tasks and reducing extra travel time is a non-trivial problem.

Customer tasks may also *vary with time*. For example, a food delivery service might receive new requests from restaurants, or an atmospheric scientist may want to update sensing locations that they submitted to a drone service provider based on prior observations. The mobility platform must handle the dynamic arrival and expiration of tasks.

3.2 Overview

Any resource-constrained system exhibits an inherent tradeoff between throughput and fairness. In the best case, the most fair schedule would also have the highest throughput; however, due to the challenges described in §3.1, it is impossible to realize this goal in many mobility settings. Mobius instead strives for customer fairness with the best possible platform throughput; its approach is to trade some short-term fairness for a boost in throughput,

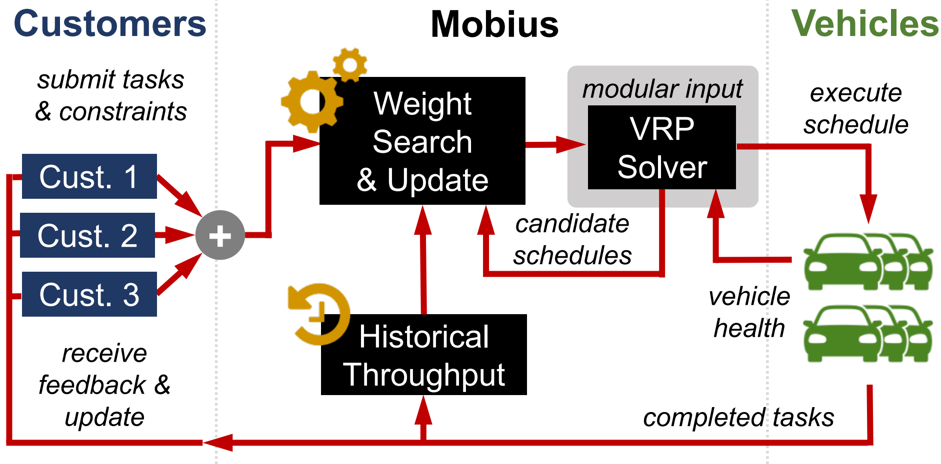


Figure 3-3: In each round, Mobius uses a VRP solver to compute a schedule that maximizes a weighted sum of throughputs, and automatically adjusts the weights across rounds to improve fairness.

while improving fairness over a longer timescale.

In each round τ, Mobius uses a VRP solver to maximize a weighted sum of customer throughputs $x_k(\tau)$.[2] Mobius sets the weights in each round to find a high throughput schedule that is approximately fair in that round. By accounting for the long-term throughputs $\overline{x}_k(t)$ delivered to each customer k in prior rounds, it is able to equalize $\overline{x}_k(t)$ over multiple rounds. We formalize this notion of balancing high throughput with fairness in §3.3. Mobius uses an iterative search algorithm requiring multiple invocations of a VRP solver to find a schedule that strikes the appropriate balance.

Our approach of trading off short-term fairness for throughput and longer-term fairness raises a natural question: why not directly schedule over a longer time horizon, rather than dividing the scheduling problem into rounds? Scheduling in rounds is desirable for several reasons: (i) their duration can correlate with the fuel or battery constraints of the vehicles, (ii) it provides a target timescale at which Mobius strives to provide fairness, (iii) shorter timescales make the NP-hard VRP problem more tractable to solve, and (iv) it enables Mobius to adapt to temporal variations in customer demand that are captured at the beginning of each round.

Fig. 3-3 shows the architecture of Mobius. In each round, customers update their task

[2] We formally define the VRP in §3.4.

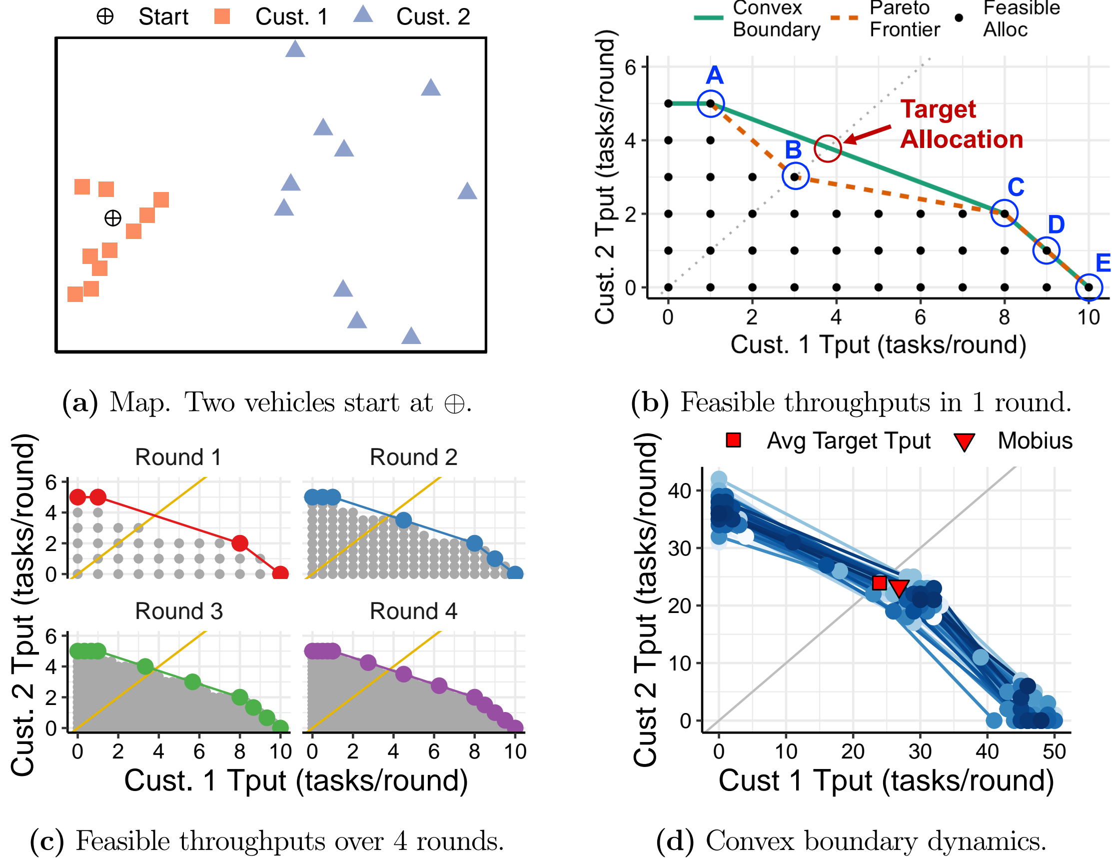


Figure 3-4: Visualizing feasible allocations of throughput for a small problem with two customers and two vehicles. Allocations on the convex boundary trade short-term fairness for throughput. The convex boundary becomes denser over time, making the target allocation achievable.

requests. Mobius then computes the best weights, generates a schedule, and dispatches the vehicles. At the end of the round, Mobius updates each customer's throughput, $\overline{x}_k(t)$, and uses this information to select weights in the next round.

3.3 Balancing Throughput and Fairness

We now provide the intuition behind our approach for balancing throughput and fairness using the example shown in Fig. 3-4. There are two customers, each requesting tasks from distributions shown on the map in Fig. 3-4a. We have two vehicles, each starting at $\oplus$. For simplicity, in §3.3.1, we consider planning schedules in 10-minute rounds, where the vehicles

return to their start locations after 10 minutes. We renew all tasks at the beginning of each round trip. Then, in §3.3.2, we explain how Mobius generalizes to dynamic settings where customer tasks change with time, and vehicles do not need to return to their starting locations.

3.3.1 Scheduling on the Convex Boundary

Feasible allocations

We first consider the set of schedules that are feasible within the time constraint. Fig. 3-4b shows the tradeoff between throughput and fairness amongst these feasible schedules. Each dot represents an allocation produced by a feasible schedule; the coordinates of the dot indicate the throughputs of the respective customers. We generate the schedules by solving the VRP for each possible subset of customer tasks.[3] We also indicate the $y=x$ line (dotted gray), which corresponds to fair allocations that give equal throughput to each customer. Note that in this example both vehicles can more easily service Customer 1. Hence, an allocation that maximizes total throughput without regard to fairness (labeled C) favors Customer 1.

Pareto frontier of feasible allocations

The Pareto frontier over all feasible allocations is denoted by the dashed orange line, containing A, B, C, D, and E. If an allocation on the Pareto Frontier achieves throughputs of x_1 and x_2 for Customers 1 and 2 respectively, there exists no feasible allocation ($\hat{x}_1$,$\hat{x}_2$) such that $\hat{x}_1 > x_1$ and $\hat{x}_2 > x_2$. The allocation that maximizes total throughput will always lie on the Pareto frontier. An allocation on the Pareto frontier is strictly superior, and therefore more desirable than other feasible allocations. So which allocation on the Pareto frontier do we pick? A strictly fair allocation lies at the point where the Pareto frontier intersects the $y=x$ line (labeled B in Fig. 3-4b). However, allocation B has low total throughput, because the vehicles spend a significant part of the 10 minutes traveling between task clusters.

[3] The VRP is NP-hard (§3.4), but because the input size is small for this example, we use Gurobi [58] to compute optimal schedules.

Convex boundary of the Pareto frontier

To capture the subset of allocations that do not significantly compromise throughput, we use the *convex boundary* of all feasible allocations, denoted by the turquoise line in Fig. 3-4b. The convex boundary is the smallest polygon around the feasible set such that no vertex bends inward [19], and the *corner points* are the vertices determining this polygon. The *target allocation* is the point where the $y = x$ lines intersects the boundary (shown in red). It has high throughput and is fair, but it may not be feasible (as in this example). Is it still possible to achieve the target throughput in such cases?

Scheduling over multiple rounds

Our key insight is that it is possible to achieve the target allocation over multiple rounds of scheduling by selecting different feasible allocations on the convex boundary in each round. In a given round, Mobius chooses the feasible allocation on the convex boundary that best achieves our fairness criteria. In our example, it chooses allocation A in its first round. By choosing allocation A over allocation B (which achieves equal throughput), Mobius compromises on short-term fairness for a boost in throughput. However, as we discuss next, it compensates for this choice in subsequent rounds. Notice that if Mobius instead chooses B, it would not be able to recover from the resulting loss in throughput.

As we compute a 10-minute schedule for each round, the set of feasible allocations expands; this allows Mobius to compensate for any prior deviation in fairness. Fig. 3-4c illustrates how the feasible set evolves over several 10-minute rounds of planning. The feasible allocations (denoted by gray dots) possible after round T are derived from the cumulative set of tasks completed in T rounds. Notice that over the four rounds, the set of feasible allocations becomes denser, and the Pareto frontier approaches the convex boundary. Thus, the target allocation (i.e., the allocation on the convex boundary that coincides with the $y = x$ line) becomes feasible.

In summary, the key insights driving the design of Mobius are: (i) the convex boundary describes a set of allocations that trade off short-term fairness for a boost in throughput, and

(ii) the Pareto frontier approaches the convex boundary over multiple rounds of planning, making it possible to correct for unfairness over a slightly longer timescale.

3.3.2 Scheduling in Dynamic Environments

In practice, environments are more dynamic: customer tasks may not recur at the same locations, and vehicles need not return to their start locations regularly. Thus, the convex boundary may not be identical in each round. However, in practice, because (i) vehicles move continuously over space and (ii) customer tasks tend to observe spatial locality, the convex boundary does not change drastically over time.

To illustrate this, we extend the example in Fig. 3-4, by creating a map with the same densities as in Fig. 3-4a, but with 50 tasks per customer. To simulate dynamics, we create a new task for each customer every 3 minutes at a location chosen uniformly at random within a bounding box. We still consider two vehicles starting at the same location (i.e., in the middle of customer 1's cluster) and plan in 10-minute rounds. We eliminate the return-to-home constraint. In order to adapt to the customers' changing tasks, we compute new 10-minute schedules every 1 minute (i.e., 10-minute rounds slide in time by 1 minute). We run this simulation for 60 minutes.

In order to understand how these dynamics impact the convex boundary as we plan iteratively, we show in Fig. 3-4d the convex boundary of 10-minute schedules at each 1-minute replanning interval. Notice that the convex boundaries hover around a narrow band, indicating that we can still track the target throughput reliably. The red square marks the value of the average target throughput across all 50 convex boundaries; we also mark the throughput achieved by Mobius's scheduling algorithm (§3.4).

In addition to the convex boundary remaining relatively stable from one timestep to the next, this method of replanning at much quicker intervals (e.g., 1 minute) than the round duration (e.g., 10 minutes) makes Mobius resilient to uncertainty in the environment.[4]

[4] §3.6 further evaluates the effectiveness of Mobius's algorithm for dynamic, real-world customer demand.

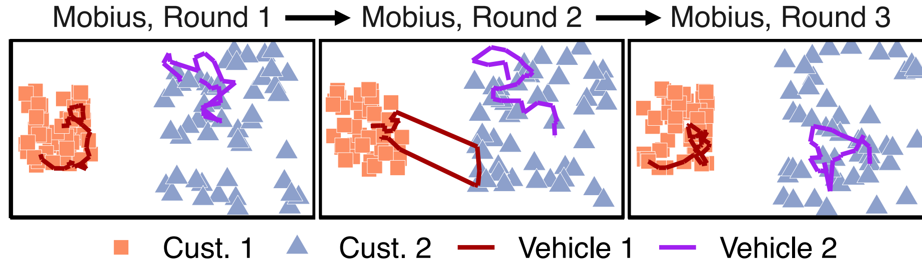


Figure 3-5: The difference in spatial density of tasks leads to short-term unfairness (Rounds 1 and 3). Mobius compensates for this by directing more resources to the underserved customer (Round 2).

For instance, Mobius can react to streaming requests in a punctual manner, and can also incorporate requests that are unfulfilled due to unexpected delays (e.g., road traffic or wind). Moreover, since Mobius uses a VRP solver as a building block to compute its schedule (§3.2), it can also leverage algorithms that solve the stochastic VRP [17], where requests arrive and disappear probabilistically.

3.3.3 Visualizing Routes Scheduled by Mobius

To illustrate how Mobius converges to fair per-customer allocations without significantly degrading platform throughput, in Fig. 3-5 we show 3 consecutive 10-minute round schedules computed by Mobius, for the dynamic example in Fig. 3-4d. In Rounds 1 and 3, we observe that Mobius decides to dedicate one vehicle to each customer in order to give them both sufficiently high throughput; here, customer 2 receives lower throughput because its tasks are more dispersed. However, in Round 2, Mobius compensates for this short-term unfairness by scheduling an additional vehicle to customer 2, while also collecting a few tasks for customer 1 in the outbound trip.

3.4 Mobius Scheduling Algorithm

Based on the insights in §3.3, we design Mobius to compute a schedule on the convex boundary in each round, such that the long-term throughputs $\overline{x}_k(t)$ approach the target

allocation. Mobius works in two steps:

1. In each round, Mobius finds the *support allocations*, which we define as the corner points on the convex boundary of the current round, near the target allocation (§3.4.1). For example, in Fig. 3-4b, Mobius would find support allocations A and C.
2. Amongst the support allocations found in step (1), Mobius selects the one that *steers the long-term throughputs* $\overline{x}_k(t)$ toward the target allocation (§3.4.2).

In this section, we present Mobius in the context of strict fairness (i.e., $\overline{x}_k(t)$ must lie along the $y{=}x$ line). §3.4.3 provides a theoretical analysis of Mobius's optimality under simplifying assumptions, and §3.4.4 describes our implementation. In §3.5, we extend Mobius's formulation to work with a class of fairness objectives.

3.4.1 Finding Support Allocations

Since the convex boundary of the Pareto frontier is equivalent to the convex boundary of the feasible set of schedules, a naive way to find the support allocations is to compute the Pareto frontier, take its convex boundary, and then identify the support allocations near the target allocation. However, computing the Pareto frontier is computationally expensive because it requires invoking an NP-hard solver an exponential number of times in the number of tasks. Mobius uses a VRP solver as a building block to find a subset of the corner points of the convex boundary around the target allocation.

The VRP involves computing a path $\mathcal{P}_v$ for each vehicle v, defined as an ordered list of tasks from the set of all tasks $\{T_k(\tau)\,|\,k{\in}K\}$, such that the time to complete $\mathcal{P}_v$ does not exceed the total time budget B for a round. VRP solvers maximize the platform throughput without regard to fairness.

We capture different priorities amongst customer tasks by assigning a weight w_k to each customer k's tasks. Let $\mathbf{x}{\in}\mathbb{R}^{|K|}$ represent a throughput vector, where x_k is the throughput for customer k, and let $\mathbf{w}{\in}\mathbb{R}^{|K|}$ represent a weight vector, with a weight w_k for each customer k.[5]

[5] $\mathbf{x}$ and $\mathbf{w}$ vary with each round τ. We drop the round index τ whenever there is no ambiguity about the

The weighted VRP seeks to maximize the total *weighted throughput* of the system, where each task is allowed a weight. We can describe this as a mixed-integer linear program:

$$\begin{aligned}
\underset{\mathcal{P}_v, \forall v \in V}{\operatorname{argmax}} \quad & \sum_{k \in K} w_k x_k = \underset{\mathcal{P}_v, \forall v \in V}{\operatorname{argmax}} \mathbf{w}^\intercal \mathbf{x} && && (3.1) \\
\text{s.t.} \quad & c(\mathcal{P}_v) \leq B && \forall v \in V && (3.2) \\
& \mathcal{P}_v \text{ is a valid path} && \forall v \in V, && (3.3)
\end{aligned}$$

where $c(\cdot)$ specifies the time to complete a path. Equation (3.2) enforces that, for each vehicle, the time to execute the selected path does not exceed the budget. Equation (3.3) captures constraints that are specific to the vehicles (e.g., if vehicles must return to home at the end of each round) and customers (e.g., if tasks are only valid during specific windows during the scheduling horizon). The weighted VRP (also called the prize-collecting VRP) is NP-hard, but there are several known algorithms with optimality bounds [8, 113].

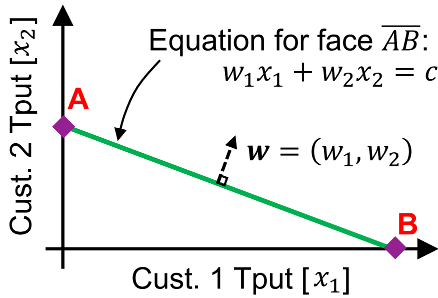
Using weights to find the corner points

We can adjust the weight vector $\mathbf{w}$ in order to capture a bias toward a particular customer; $\mathbf{w}$ describes a direction in the customer throughput space, reflecting that bias. Fig. 3-6a visualizes $\mathbf{w}$ in a 2-D customer throughput space. A solver optimizing for Equation (3.1) searches for the schedule with the highest throughput in the direction of $\mathbf{w}$ [20], thus requiring the schedule to lie on the convex boundary. For example, $\mathbf{w_1} = (1,0)$ finds the schedule on the convex boundary that prioritizes customer 1 (i.e., along the x-axis), and $\mathbf{w_2} = (0,1)$ finds a schedule that prioritizes customer 2 (i.e., along the y-axis).

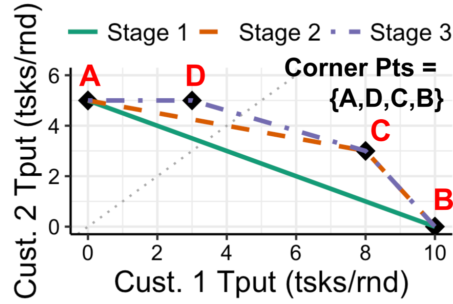
Searching on the convex boundary

Recall that, for strict fairness, the target allocation is the point where the $y = x$ line intersects the convex boundary for the current round (§3.3). At the start of the search, Mobius does not

current round.



(a) Computing **w** for a face.



(b) Finding support allocations.

Figure 3-6: Using a blackbox VRP solver as a building block, Mobius runs an iterative search algorithm to find the support allocations.

yet know the convex boundary, so it cannot know the target allocation. To find allocations on the convex boundary, Mobius employs an iterative search algorithm, analogous to binary search; in each stage, it tries to find a new allocation on the convex boundary in the direction of the $y = x$ line. Mobius begins the search with allocations along the customer axes. For two customers, it begins with weights $\mathbf{w_1}$ and $\mathbf{w_2}$ above, which gives two allocations on the convex boundary. In each stage of the search, Mobius computes a new weight vector, using allocations found on the convex boundary in the previous stage, in order to find a new allocation on the convex boundary. It terminates when no new allocation can be found. By searching in the right direction, Mobius only needs to compute a subset of corner points on the convex boundary.

To better illustrate the algorithm, consider the example in Fig. 3-6b, with 2 customers. Mobius starts the search by looking at the two extreme points on the customer 1 (x_1) and customer 2 (x_2) axes, which correspond to prioritizing all vehicles for either customer. So in stage 1, Mobius computes these schedules, using the weight vectors $\mathbf{w_1} = (1{,}0)$ and $\mathbf{w_2} = (0{,}1)$, which give the allocations A and B, respectively, in Fig. 3-6b. After stage 1, $\{A{,}B\}$ is the current set of corner points determining the convex boundary.

In the next stage, Mobius computes a new weight $\mathbf{w}$ to continue the search in the direction normal to $\overline{AB}$ (Fig. 3-6a). Let the equation for the face $\overline{AB}$ be $w_1x_1 + w_2x_2 = c$, where w_1, w_2, and c can be derived using the known solutions on the line, A and B. So, by

invoking the VRP solver (Equation (3.1)) with $\mathbf{w}=(w_1,w_2)$, we try to find a schedule on the convex boundary, with the highest throughput in the direction normal to $\overline{AB}$. Let $\hat{x}_1$ and $\hat{x}_2$ be the throughputs for the schedule computed with weight $\mathbf{w}$. If $(\hat{x}_1,\hat{x}_2)$ lies above this line, i.e., $w_1\hat{x}_1+w_2\hat{x}_2>c$, then the point $(\hat{x}_1,\hat{x}_2)$ is a valid extension to the convex boundary. In this example, Mobius finds a new allocation C; so, the new set of corner points is $\{A,C,B\}$.

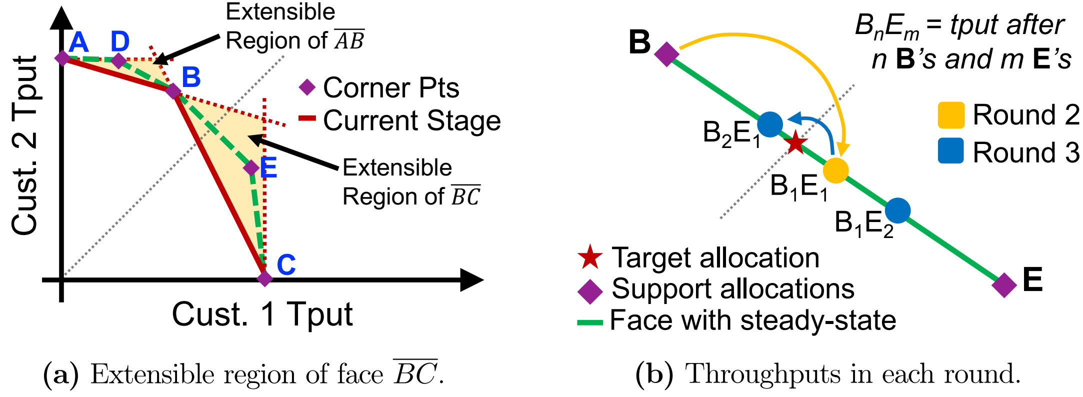
Notice that this extension in stage 2 creates two new faces on the convex boundary, $\overline{AC}$ and $\overline{CB}$. But, the $y=x$ line only passes through $\overline{AC}$. So, in stage 3, Mobius continues the search, extending $\overline{AC}$ by the computing the weights as described above (normal to $\overline{AC}$), and discovers a new allocation D. Finally, Mobius tries to extend the face $\overline{DC}$ because it intersects the $y=x$ line. It finds no valid extension, and so, it terminates its search on the face $\overline{DC}$, and returns the support allocations D and C.

Generalizing to more customers

Mobius computes a weight for each customer $k\in K$, i.e., $\mathbf{w}\in\mathbb{R}^{|K|}$. Faces on the convex boundary become $|K|$-dimensional hyperplanes, described by the equation $\sum_{k\in K}w_kx_k=c$. Mobius solves for $\mathbf{w}$ using the $|K|$ allocations that define each face, and finds $|K|$ support allocations at the end of the search. Recall from the example in §3.4.1 that each stage produced 2 new faces and that Mobius only continued the search by extending 1 face. With $|K|$ customers, even with $|K|$ new faces after each stage, Mobius only invokes the VRP solver once to continue the search. A naive algorithm, by contrast, would require $|K|$ calls to the VRP solver in each stage. Thus Mobius scales easily with more customers by pruning the search space efficiently.

3.4.2 Scheduling Over Rounds

In each round, Mobius finds $|K|$ support allocations, which determine the face of the convex boundary that contains the target allocation. It then selects a support allocation among these $|K|$ such that the per-customer long-term throughputs $\overline{x}_k(t)$ approach the target throughput.



(a) Extensible region of face $\overline{BC}$.

(b) Throughputs in each round.

Figure 3-7: Mobius (a) finds the support allocations nearest the target allocation in each round, and (b) converges to the target allocation.

By tracking $\overline{x}_k(t)$ over many rounds, Mobius can select allocations that compensate for any unwanted bias introduced to some customer in a prior round.

Mathematically, to choose a schedule in round t, Mobius considers the effect of each support allocation $\mathbf{x}(t)$ on the average throughput $\overline{\mathbf{x}}(t+1)$. The average throughput is defined for each customer k as $\overline{x}_k(t+1) = \gamma_t x_k(t) + (1-\gamma_t)\overline{x}_k(t)$, where $\gamma_t = 1/(t+1)$. Mobius chooses $\mathbf{x}(t)$ such that $\overline{\mathbf{x}}(t+1)$ is closest to the $y = x$ line (in Euclidean distance).

3.4.3 Optimality of Mobius

Mobius is optimal in a round

We can prove that Mobius finds the support allocations nearest the target throughput (in Euclidean distance). We illustrate this through the example in Fig. 3-7a, where the corner points of the convex boundary are $\{A, D, B, E, C\}$, and B is closest to the target allocation. In the previous stage, Mobius discovered B, and it needs to pick one face to continue the search. The shaded yellow regions indicate the extensible regions of the two candidate faces $\overline{AB}$ and $\overline{BC}$. The extensible region of a face describes the space of allocations that can be obtained by searching with the weight vector that defines that face, while maintaining a convex boundary (§3.4.1). Since Mobius finds a new allocation on the convex boundary in every stage of the search, no allocation can exist outside these regions; otherwise, the resulting set of discovered

allocations would no longer be convex. Thus, the best face for Mobius to continue the search is indeed $\overline{BC}$, because its extensible region is the only one that may contain a better allocation closer to the $y = x$ line. App. A.3.1 includes a formal proof that the optimal support allocation (i.e., the allocation closest to the line $y = x$) is unique and that Mobius finds it.

Optimality over multiple rounds

Under a static task arrival model, we can show that the schedules computed by Mobius achieve throughputs that are optimal at the end of every round, i.e., the achieved throughput has the minimum distance possible to the target allocation after each round. This model assumes the convex boundary remains the same across rounds. One way to realize this is to require (i) the vehicles return to their starting locations at the end of each round, and (ii) all tasks are renewed at the beginning of each round. We make these simplifying assumptions only for ease of analysis; our evaluation in §3.6 does not use them.

We describe an intuition for this result below. [6] Per the static task arrival model, the convex boundary is the same in *each subsequent round*; therefore, Mobius finds the same support allocations in every round. By taking into account the long-term per-customer rates, $\overline{x}_k(t)$, Mobius oscillates between these support allocations in each round at the right frequency, such that $\overline{x}_k(t) \forall k \in K$ converges to the target allocation over multiple rounds. We illustrate this in Fig. 3-7b, which shows the support allocations B and E. The face $\overline{BE}$ contains the target allocation, denoted by the star. Because Mobius oscillates between B and E, the allocation $(\overline{x}_1(t), \overline{x}_2(t))$ must lie along $\overline{BE}$. Mobius chooses B in the first round because its throughput is closer than E to the target allocation. In the second round, it chooses E, moving the average throughput to B_1E_1. In the third round, Mobius chooses B, moving the average throughput to B_2E_1. Notice that if it had instead chosen E in the third round, the average throughput would be B_1E_2, which is further away from the target throughput. Thus, this myopic choice between B and E results in the closest solution to the target allocation after any number of

[6] See App. A.3.2 for a formal proof.

rounds. Additionally, notice that the length of the jump (e.g., from B to B_1E_1 and from B_1E_1 to B_2E_1) decreases in each round; therefore, Mobius *converges* to the target throughput.

3.4.4 Implementation

We implement the core Mobius scheduling system in 2,300 lines of Go.[7] It plugs directly with external VRP solvers implemented in Python or C++ [58, 98]. Mobius exposes a simple, versatile interface to customers, which we call an interest map. An interest map consists of a list of desired tasks, where each task includes a geographical location, the time to complete the task once the vehicle has reached the location, and a task deadline (if applicable). In each round, Mobius gathers and merges interest maps from all customers, before computing a schedule. At the end of each round, it informs the customers of the tasks that have been completed, and customers can submit updated interest maps. Interest maps serve as an abstraction for Mobius to ingest and aggregate customer requests; however, the merged interest map is directly compatible with standard weighted VRP formulations [8, 44] without modification. Thus, Mobius acts as an interface between customers and vehicles, using a VRP solver as a primitive in its scheduling framework (Fig. 3-3).

Bootstrapping VRP solvers

Since the VRP is NP-hard [113], solvers resort to heuristics to optimize Equation (3.1). In practice, we find that state-of-the-art solvers do not compute optimal solutions; however, we can aid these solvers with initial schedules that the heuristics can improve upon. We warm-start the VRP solvers with initial schedules generated by the following policies: (i) maximizing throughput, (ii) dedicating vehicles (assuming a sufficient number of vehicles), and (iii) a greedy heuristic that maximizes our utility function (§3.5). [8] At the beginning of each round, Mobius builds a suite of warm start solutions. Then, prior to invoking the VRP solver with some weight vector $\mathbf{w}$, Mobius chooses the initial schedule from its warm start

[7] github.com/mobius-scheduler/mobius

[8] App. A.4 describes this heuristic in detail.

suite with the highest weighted throughput (i.e., objective of Equation (3.1)). Mobius also caches the schedules found from all invocations to the VRP solver (§3.4.1), to use for warm start throughout the round. Mobius parallelizes all independent calls to the VRP solver (e.g., when computing warm start schedules and when generating $|K|$ schedules to initialize the search along the convex boundary).

3.5 Generalizing to α-Fairness

The fairness objective we have considered so far aims to provide all customers with the same long-term throughput (maximizing the minimum throughput). However, an operator of a mobility platform may be willing to slightly relax their preference for fairness for a boost in throughput. To navigate throughput-fairness tradeoffs, we can generalize Mobius's algorithm (§3.4) to optimize for a general class of fairness objectives. We use the α-parametrized family of utility functions U_α, developed originally to characterize fairness in computer networks [74]:

$$U_\alpha(\mathbf{y}) = \sum_{k \in K} \frac{{y_k}^{1-\alpha}}{1-\alpha}, \tag{3.4}$$

where $\mathbf{y} \in \mathbb{R}^{|K|}$ and y_k is the throughput of customer k (either short-term x_k or long-term $\overline{x}_k$). U_α captures a general class of concave utility functions, where $\alpha \in \mathbb{R}_{\geq 0}$ controls the degree of fairness. For instance, when $\alpha = 0$, the utility simplifies to the throughput-maximizing objective defined in Equation 3.1 (assuming all customers have the same weight). By contrast, when $\alpha \to \infty$, the objective becomes maximizing the minimum customer's throughput (i.e., max-min fairness). $\alpha = 1$[9] corresponds to proportional fairness, where the sum of log-throughputs of all customers is maximized; this ensures that no individual customer's throughput is completely starved.

[9] U_α is not defined at $\alpha = 1$, so we take the limit as $\alpha \to 1$.

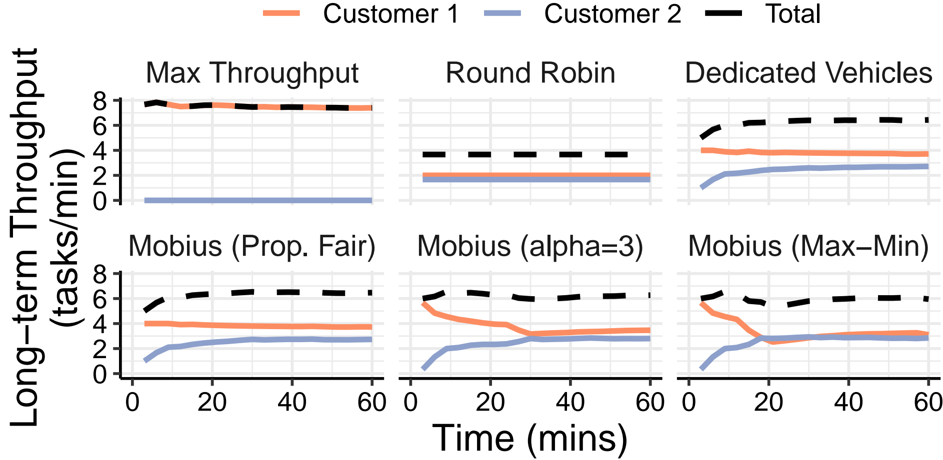


Figure 3-8: Mobius can tune its allocation to deliver proportional fairness ($\alpha = 1$) and max-min fairness (approximated with $\alpha = 100$).

Generalizing Mobius's search algorithm

When Mobius generalizes to α-fairness, the target allocation is no longer simply the point on the convex boundary that intersects the $y = x$ line. The target allocation is instead the allocation on the convex boundary with the greatest utility U_α. When searching the convex boundary in each round, Mobius determines which candidate face contains the target throughput by using Lagrange Multipliers to find the point *along the face* [10] with the greatest utility. Once it finds each support allocation $\mathbf{x}$, Mobius incorporates the historical throughput $\overline{\mathbf{x}}$ to select the schedule with greatest cumulative utility $U_\alpha(\gamma_t \mathbf{x}(t+1) + (1-\gamma_t)\overline{\mathbf{x}}(t))$, where γ_t is as defined in §3.4.2.

An example

Fig. 3-8 shows a time-series chart of long-term customer and platform throughputs for the example described in §3.3.2. By adapting to different schedules on the convex boundary, Mobius converges to a fair allocation of rates without degrading total throughput. α allows Mobius to compute *expressive schedules*; for instance, $\alpha = 1$ strives to maximize total

[10] App. A.1 shows how to find the face containing the target throughput.

throughput without starving either customer. Additionally, Mobius (max-min)[11] converges to a fair allocation of long-term throughputs within 20 minutes.

3.6 Real-World Evaluation

We evaluate Mobius using trace-driven emulation (§3.6.1) in two real-world mobility platforms. In §3.6.2, we apply Mobius to Lyft ridesharing in Manhattan and demonstrate that it scales to large online problems. In §3.6.3, we deploy Mobius on a shared aerial sensing system, involving multiple apps with diverse spatiotemporal preferences. Our evaluation focuses on answering the following questions:

- How does Mobius compare to traditional approaches in online scheduling for large-scale mobility problems?
- How robust is Mobius in the presence of dynamic spatiotemporal demand from customers?
- How can we tune Mobius's timescale of fairness?
- What other benefits does Mobius provide to customers, beyond optimizing per-customer throughputs?

3.6.1 Online Trace-Driven Emulation

We implement a trace-driven emulation framework to compare Mobius against other scheduling schemes, under the same real-world environment. This framework replays timestamped traces of requests submitted by each customer, by streaming tasks to the scheduler as they arrive, and sending task results back to the customer.

[11]Mobius approximates max-min fairness ($\alpha \to \infty$) with $\alpha = 100$.

Capturing environment dynamics and uncertainty

To emulate dynamic customer demand, our emulation framework streams tasks according to the timestamps in the trace—so Mobius has no visibility into future tasks. To emulate uncertainty in customer demand, we cancel tasks that are not scheduled in 10 minutes. Additionally, the case studies in §3.6.2 and §3.6.3 consider scenarios where *at least* one customer is backlogged (defined in §3.1). If no customers are backlogged, then the platform can fulfill all tasks within the planning horizon, and the resulting schedule would have maximal throughput and fairness. Thus, the problems are only interesting when at least one customer is backlogged; Mobius is effective and required only in such situations.

Backend VRP solver

We use the Google OR-Tools package [98] as our backend weighted VRP solver (Equation (3.1)). OR-Tools is a popular package for solving combinatorial optimization problems, and supports a variety of VRP constraints, including budget, capacity, pickup/dropoff, and time windows. Our case studies involve VRPs with different sets of constraints. We run our experiments on an Amazon EC2 `c5.9xlarge` instance with 36 CPUs.

Baselines

In our experiments, we evaluate Mobius's throughput and fairness against two baseline routing algorithms: (i) a max throughput scheduler, and (ii) dedicated vehicles. The max throughput scheduler simply runs the backend VRP solver on the same input of customer tasks fed into Mobius for a round. This solution provides a benchmark on the platform capacity, and quantifies the maximum achievable total throughput. We compute the "dedicated vehicles" schedule by first distributing the vehicles evenly among all customers,[12] and then invoking the max throughput scheduler once for each customer. This solution provides a benchmark schedule that divides vehicle time equally among all customers. As shown in §3.1, round-robin

[12] Dedicating vehicles is most suitable when the number of vehicles is a multiple of the number of customers.

scheduling achieves very low throughput; hence we omit it from the results in this section.

To the best of our knowledge, Mobius is the first algorithm that explicitly optimizes for customer fairness in mobility platforms. We considered evaluating Mobius by running a scheduler that optimizes throughput and fairness over a longer timescale using a mixed-integer linear program solver (e.g., Gurobi [58] or CPLEX [65]); however, this is not feasible in practice, because (i) customer demands arrive in a streaming fashion, and (ii) these solvers do not scale beyond tens of tasks [89]. Thus, we believe the baselines described above offer reasonable comparisons for Mobius.

Microbenchmarks

In addition to the real-world case studies (§3.6.2-§3.6.3), we also evaluate Mobius on microbenchmarks created from synthetic customer demand, including scenarios where Mobius is optimal (under the static task arrival model, §3.4.3). We compare Mobius against max throughput, dedicating vehicles, and round robin, and show, through controlled experiments, that (i) it provides provably good throughput and fairness for a variety of spatial demand patterns, (ii) it scales for different numbers of vehicles, (iii) it controls its timescale of fairness, and (iv) it can tune its fairness parameter α. We also report the runtime of Mobius in various environments. We include these results in App. A.5 and App. A.6.

3.6.2 Case Study: Lyft Ridesharing in Manhattan

Motivated by the issue of "destination discrimination" [88, 116, 131] discussed at the beginning of this chapter, we consider a ridesharing service that receives requests from different neighborhoods (customers) in a large metro area. Some neighborhoods are easier to travel to than others, and rider demand out of a neighborhood can vary with the time of day. We show that Mobius can guarantee a fair quality-of-service (in terms of max-min fair task fulfillment) to all neighborhoods throughout the course of a day, without significantly compromising throughput. We also show that, although it optimizes for an equal allocation

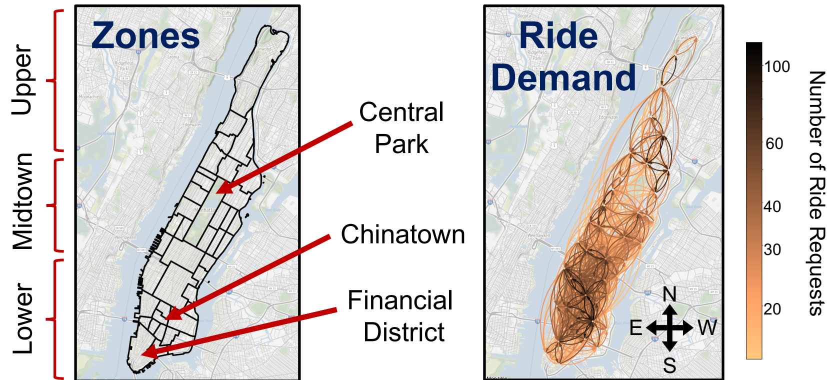


Figure 3-9: Maps of zones (customers) and demand in Manhattan, indicating skews in both spatial coverage and volume of ride requests.

of throughputs, Mobius does not degrade other quality-of-experience metrics, such as rider wait times. We further demonstrate that Mobius is a scalable *online* platform that generates schedules for a large city-scale problem.

Ridesharing demand

We use a 13-hour trace of 16,817 timestamped Lyft ride requests, published by the New York City Taxi and Limousine Commission, involving 40 neighborhoods (zones) in Manhattan over the course of a day [33]. Each request consists of a pickup and a dropoff zone, and we seek to provide pickups from all zones equitably. The map in Fig. 3-9 (left) demarcates the customer zones.

Fig. 3-9 (right) illustrates the scale of this scheduling problem. It visualizes traffic on the top 1,000 (out of 3,300) pickup-dropoff pairs; the color of each arrow indicates the volume of ride requests for that pickup-dropoff location. Notice that both the distance of rides and the volume of requests originating from zones vary vastly throughout the island. A significant fraction of requests arrive into and depart from Lower Manhattan. Some zones in Upper Manhattan have as few as 15 unique outbound trajectories, while other zones have hundreds.

Moreover, ridesharing demand varies significantly with the time of day. For instance, a busy zone near Midtown Manhattan sees the load vary from around 200 to 600 requests/hour, and a quiet zone near Central Park experiences a minimum load of 3 requests/hour and

peak load of 24 requests/hour. Notice that the dynamic range of demand throughout the 13 hours also varies across zones.

Experiment setup

This ridesharing problem maps to the capacitated pickup/delivery VRP formulation [44]. It computes schedules that maximize the total number of completed rides, such that (i) a ride's pickup and dropoff are completed on the same vehicle, and (ii) each vehicle is completing at most one ride request at any point in time. We configure the solver to retrieve real-time traffic-aware travel time estimates from the Google Maps API [55], and we constrain OR-Tools to report a solution within 3 minutes.

We use the trace described above in our emulation framework (§3.6.1). We compute schedules for a fleet of 200 vehicles.[13] In order to ensure that the schedules are not myopic, we plan our routes with 45-minute horizons; however, to reduce rider wait times, we recompute the schedule every 10 minutes, while ensuring that we honor any requests that we have already committed to in the schedule. We assume that riders cancel requests that are not incorporated into a schedule within 10 minutes of the request time.

Fairness with high vehicle utilization

Since Mobius plans continuously, having several allocations on the convex boundary at its disposal, we expect it to converge to a fair allocation of rates, despite the skew in demand. Fig. 3-10 shows the long-term throughputs achieved for each zone by different scheduling algorithms, after 13 hours. The color of each zone in the map indicates that zone's throughput. Bright colors correspond to high throughput, and a homogeneous mix of colors indicates a fair allocation. Beneath the maps, we also stack the zone throughputs to indicate how each scheduler divides up the total platform throughput across the zones; ideally we would like large, evenly-sized blocks.

[13] The number of vehicles does not matter, since we compare Mobius to the platform capacity (from the max throughput scheduler).

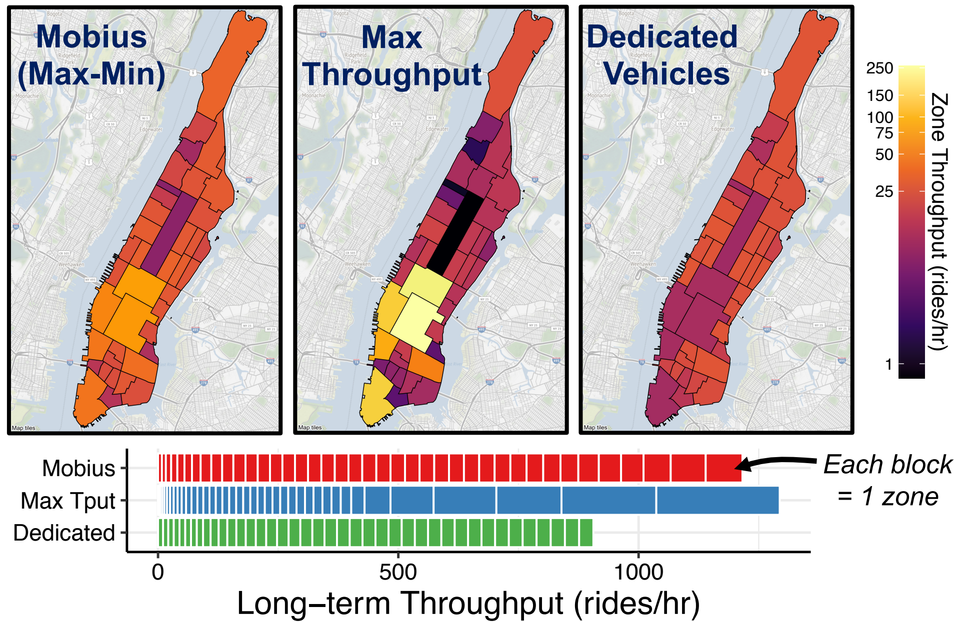


Figure 3-10: Long-term throughputs for zones in Manhattan after 13 hours. A good scheduler should have a stacked plot with large evenly-sized blocks, and a map with bright (high throughput) and homogeneous (fair) colors across zones.

The max throughput scheduler divides the platform throughput most unevenly across zones. In particular, we see that while it serves nearly 200 rides/hour out of the Financial District (Lower Manhattan), it virtually starves zones near Central Park. From the demand map (Fig. 3-9), notice that (i) a majority of rides originate from Lower Manhattan, and (ii) most of these trips are destined for neighboring zones. Thus, the best policy to maximize the total number of trips completed is to stay in Lower Manhattan, which is what the max throughput scheduler does.

The bar chart indicates that dedicating 5 vehicles to each zone results in 40% lower platform throughput than the max throughput scheduler. This is because a heterogeneous demand across zones cannot be effectively satisfied by an equal division of resources (vehicles). Nevertheless, Fig. 3-10 shows that this scheduler shares the platform throughput most evenly across zones. The division of per-zone throughputs is not perfectly even, in spite of dedicating an equal number of vehicles, because (i) ride requests from different zones can have different trip lengths, and (ii) some zones have inherently low demand and do not backlog the system,

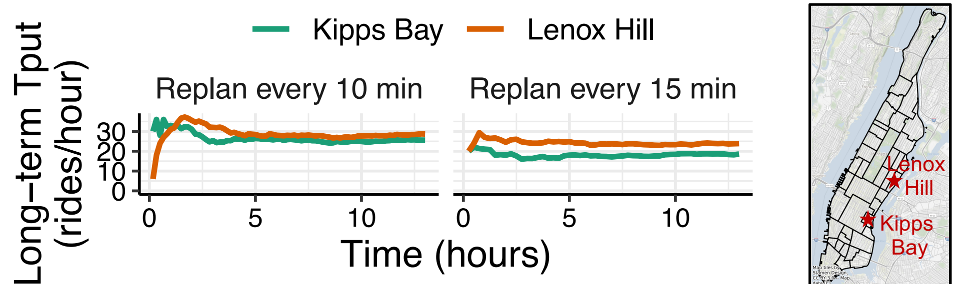


Figure 3-11: Time series of long-term throughputs for two zones for different replanning horizons. Frequent replanning ensures fairness (equal throughputs) at shorter timescales.

leaving some vehicles idle.

By contrast, Mobius strikes the best balance between throughput and fairness. It achieves roughly equal zone throughputs, while compromising only 10% of the maximum platform throughput. Compared to dedicating vehicles, we see, from the map, that Mobius achieves higher throughput for most zones by identifying an incentive to chain requests from different zones. For example, Mobius combines two requests from different zones into the same trip, when the dropoff of the first request is close to the pickup of the second request. While this helps improve efficiency, Mobius also prioritizes pickups from zones with a historically low throughput to ensure fairness across zones. This ridesharing simulation reveals that it is possible to achieve a fair allocation of rates in a practical setting *without* significantly degrading platform throughput.

Controlling the timescale of fairness

Mobius's replanning interval controls the timescale over which it is fair. The more often that Mobius replans, the more up-to-date its record of long-term customer throughputs; Mobius can then adapt to short-term unfairness quickly by finding a more suitable schedule on the convex boundary. Recall that when replanning frequently, the convex boundary does not change drastically between scheduling intervals (§3.3.2), if the spatial distribution of tasks do not change rapidly with time. So, in practice, we do not expect to deviate far from the ideal target throughput. Fig. 3-11 shows the long-term throughputs achieved for two zones, for

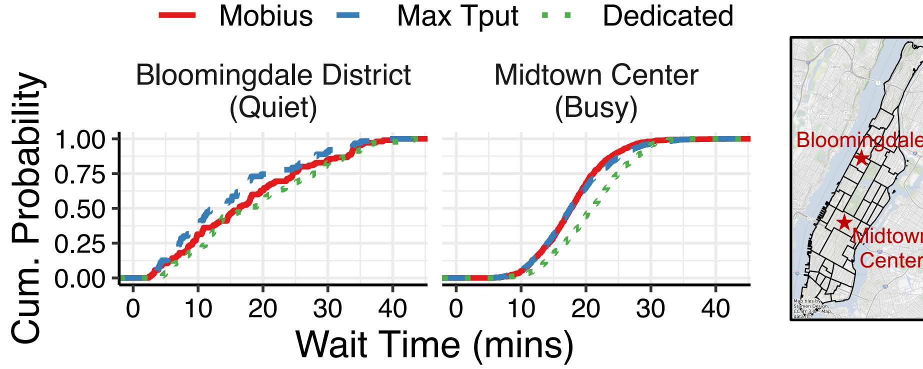


Figure 3-12: Distributions of rider wait times for two zones. Even though Mobius compromises some throughput for fairness, it delivers similar wait times as the max throughput scheduler.

Traffic	Parking	Air Quality	iPerf	Roof
7.2 m/s, 5.2 m/s, 5.6 m/s	12 parked cars	PM 2.5 sensor; Measured PM 2.5	TCP Tput (Mbps): 5, 10, 15, 20, 25	
Measure average vehicle speed.	Count occupied spots in parking lot.	Map air quality of plume (AQI).	Profile cellular connectivity.	Image residential roofs.
• 11 continuous monitoring tasks (10 sec/task) • Prioritizes tasks with high variance in speed	• 3 recurring tasks (60 sec/task) • Tasks renew after 10 mins	• 50 one-time tasks (20 sec/task) • Prioritizes using Gaussian Process model	• 100 cyclic monitoring tasks (10 sec/task) • Renews all tasks after each cycle	• 60 one-time tasks (20 sec/task) • No prioritization among tasks

Figure 3-13: Summary of aerial sensing applications, which span a variety of spatial demand and reactive/continuous sensing preferences. We collected ground truth data for each of these applications using real drones, and created traces to evaluate Mobius.

replanning timescales of 10 minutes and 15 minutes. Mobius equalizes throughputs better when it replans more frequently.

Rider wait times

Platform operators prefer high throughput schedules because they translate directly to high revenue; low throughput would lead to more cancelled rides. While Fig. 3-10 demonstrates that Mobius is fair without degrading throughput, we would like to know if optimizing for fairness impacts rider wait time (i.e., the time between requesting a ride and being picked up).

Fig. 3-12 compares the distributions of rider wait times for rides originating from Bloomingdale District (a quiet neighborhood west of Central Park) and from Midtown Center (a busy district near Times Square). We compute wait times are only for fulfilled tasks. Notice that in both zones—with two very different demand patterns—the distribution

of wait times for Mobius is comparable to that of the max throughput scheduler.

We observe that the wait times in the quiet zone are slightly higher for Mobius (average of 17 minutes, compared with 15 minutes for max throughput). This is because the wait times for Mobius are computed for significantly more tasks (Mobius fulfills 66.7% more ride requests than does max throughput). The schedule that dedicates vehicles sees higher wait times than Mobius, especially when rides originate from a busy zone (e.g., Midtown Center), since vehicles would be idle until they return to their assigned zone to pick up a new rider.

Scalability

This case study demonstrates that Mobius is practical at an urban scale. In fact, when scheduling its fleet of taxis, New York City's Yellow Cab restricts its scheduling region to Manhattan and organizes its requests according to approximately 40 taxi zones [16, 32]. In our experiments, the backend VRP solver (i.e., max throughput scheduler) computes each 45-minute schedule in 3 minutes (capped by the timeout). We observe that Mobius takes 5-6 minutes; Mobius sees a speedup by (i) parallelizing calls to the VRP solver and (ii) warm-starting the VRP solver with initial schedules (§3.4.4). These optimizations help Mobius easily scale to tens of thousands of tasks. We believe we can further improve the speed by leveraging parallelism in the backend VRP solver [111] (OR-Tools does not expose a multi-threaded solver).

3.6.3 Case Study: Shared Aerial Sensing Platform

The recent proliferation of commodity drones has generated an increased interest in the development of aerial sensing and data collection applications [3, 6, 39, 46, 85, 86], as well as general-purpose drone orchestration platforms [59, 90, 99]. An emerging mobility platform is a drones-as-a-service system [47, 60, 84, 117, 124], where developers submit apps to a platform that deploys these app tasks on a shared fleet of drones. App (customer) semantics in a drone sensing platform can show significant heterogeneity in both space and time. To ensure a satisfactory QoS for all applications, a scheduler must not only efficiently multiplex tasks from

Figure 3-14: Map of tasks for 5 aerial sensing apps, spanning a 1 square mile area in Cambridge, MA. Mobius replans every 5 minutes, in order to incorporate new requests. Each drone returns to recharge every 15 minutes.

different applications in each flight (typically constrained to 20 minutes due to the battery life [41]), but also share task completion throughput equitably across apps. Since apps can be reactive (i.e., sensing preferences change as apps receive measured data), Mobius must additionally provide a sustained rate-of-progress to each app, as opposed to "bursty" throughput.

Sensing apps

We implement 5 popular urban sensing apps to evaluate Mobius in this drones-as-a-service context, summarized in Fig. 3-13. Fig. 3-14 depicts the locations for the sensing tasks submitted by each app. We describe each app below:

- The *Traffic app* continuously monitors road traffic congestion over 11 contiguous segments of road in an urban area. To measure average vehicle speed, it collects 10-second video clips at each road segment, detects all cars using YOLOv3 [104], and tracks the trajectory [21] of each vehicle. After gathering multiple initial samples at all 11 locations, the app prioritizes the locations with the highest variance in speed, in order to collapse uncertainty in its overall estimates of road congestion.
- The *Parking app* counts parked cars at 3 sites, by monitoring each lot for 1 minute; to maintain fresh estimates of counts, this app renews these 3 tasks after 10 minutes.

- The *Air Quality app* measures PM2.5 concentration around a plume [1], submitting a candidate list of 100 one-time sampling locations. This app is also reactive; on receiving a measurement, it updates a Gaussian Process model [102] and cancels any unfulfilled tasks with high predicted accuracy.
- The *iPerf app* builds a map of cellular coverage in the air, by profiling throughput at 100 spatially-dispersed locations. It renews all tasks after each cycle of 100 measurements is complete.
- The *Roof app* submits 60 one-time tasks to image roofs over a residential area.

Notice that these apps collectively have a variety of spatiotemporal characteristics. For instance, the Traffic app changes its requests with time, based on the uncertainty in speed estimates and the freshness in measurements. By contrast, the Air Quality app changes its requests with space, using a statistical model to collapse uncertainty in a task based on nearby measurements. The iPerf app has no temporal preferences, and instead functions as a "free-riding" app that gathers quick measurements over a large area.

Ground-truth data collection

To run our drones-as-a-service platform on real-world sensor data, which is critical to the performance of the reactive and continuous monitoring apps, we separately gather 90 minutes of ground-truth data for each app, using real drones. This gives us a trace of timestamped measurement values of each app. We then use our trace-driven emulation framework (§3.6.1) to evaluate different scheduling algorithms. Fig. 3-13 shows highlights from our data collection. For example, to collect ground-truth for the Traffic app, we instrument 6 DJI Mavic Pros [41] to continuously gather video and track cars over the 11 measurement locations (Fig. 3-14) for 90 minutes. Similarly, for the iPerf and Air Quality apps, we program a DJI F450 drone [42] equipped with an LTE dongle and a PM2.5 sensor [1] to gather measurements at their respective measurement locations. We instrument our drone to communicate its location, battery status, and measurement data to a dashboard hosted

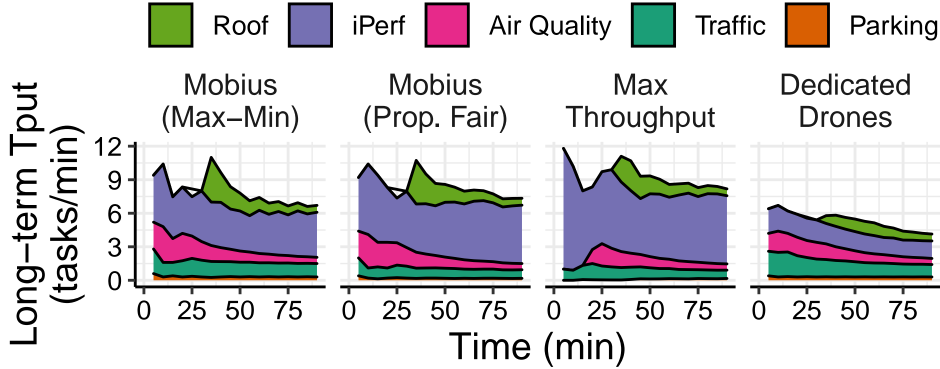


Figure 3-15: Long-term throughputs achieved over 90 minutes. Mobius achieves high throughput and best shares it amongst the apps.

on an EC2 instance, from which we observe the drone's progress on our laptop.

Experiment setup

We configure our backend solver to estimate travel time as the Euclidean distance between the sensing tasks plus the sensing time for the destination task. In order to be sufficiently reactive to the Traffic and Air Quality apps, we schedule in 5-minute rounds, and require that the drones return to recharge their batteries every 15 minutes. We run our trace-driven emulation framework with 5 drones. Additionally, we configure the Roof app to join the system after 30 minutes.

High throughput, high fairness

To understand how Mobius divides the platform throughput, we show the long-term throughput for each app over 90 minutes in Fig. 3-15. Mobius (max-min) achieves 55% more throughput than dedicating drones and only 15% less throughput than maximizing throughput. Mobius with a proportional fairness objective similarly outperforms max throughput and dedicated vehicles in navigating the throughput-fairness tradeoff. Note that the throughputs of the Air Quality and Roof apps decay with time, after their one-time tasks are fulfilled.

Because these apps have variable demand (e.g., 100 tasks for iPerf and 3 tasks for Parking), studying throughput is not sufficient. Hence, we plot the tasks completed as a

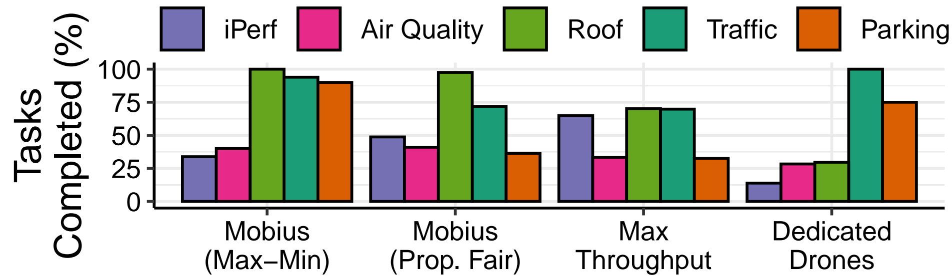


Figure 3-16: Percentage of tasks completed per app. Mobius fulfills nearly all requests for the Traffic and Parking apps, before allocating "excess" vehicle time to the more backlogged apps.

fraction of demand for each app in Fig. 3-16. Notice that, under Mobius, even the most starved app (iPerf) completes nearly 34% of its tasks; by contrast, max throughput and dedicated drones deliver worst-case task completions of 30% and 13%, respectively. Even though dedicating drones guarantees equal drone time for each app, it is extremely unfair toward apps with higher demand or more spatially-distributed tasks.

Impacts of sensing and travel times

Fig. 3-14 would suggest that the Air Quality and Roof tasks are easier to service, since their tasks are more spatially concentrated; however, their tasks take 20 seconds each (Fig. 3-13). The max throughput scheduler understands this tradeoff in terms of maximizing throughput, and thus prioritizes the iPerf app, since its 10-second tasks (Fig. 3-13) are cheap to complete. By contrast, Mobius additionally understands how to navigate this tradeoff in terms of fairness; for instance, it forgoes some iPerf tasks to complete more 20-second AQI measurements.

Reliable rate-of-progress

In enforcing either proportional or max-min fairness, Mobius does not starve any app, at any instant of time. Indeed, Fig. 3-15 indicates that Mobius delivers a reliable rate-of-progress to the Air Quality app, gradually giving it roughly 3 tasks/min over the first 20 minutes. By contrast, the max throughput scheduler is more "bursty", and only services this app after

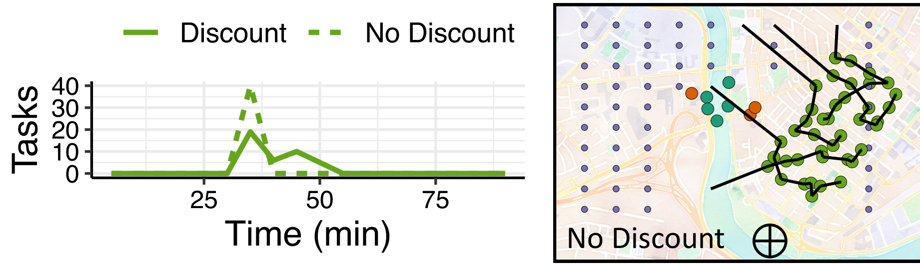


Figure 3-17: Discounting long-term throughput allows Mobius to gradually respond to the sudden presence of the transient Roof app, instead of dedicating all drones to it.

20 minutes. As a result, we find that, with Mobius, the root-mean-square error (RMSE) of the Gaussian Process model for the air quality drops more rapidly.

Catering to transient apps

Recall that the Roof app joins the platform after 30 minutes. Fig. 3-15 indicates that Mobius rapidly adapts to this change in demand with a spike in throughput for the Roof app at the cost of lower throughput for the iPerf and Air Quality apps. Notice that this spike in Mobius's schedule is larger in magnitude than the one in the max throughput schedule. This is because Mobius realizes that, when the Roof app joins, it has no accumulated throughput, while other apps have amassed higher throughput from living in the system for longer. Fig. 3-17 (right) plots the routes for all 5 drones during minutes 30-35; all drones immediately flock to the Roof app. With Mobius, an operator can choose to respond to the arrival of new apps by discounting throughput accumulated in prior rounds. Fig. 3-17 (left) shows how Mobius can control the Roof app's rate of task fulfillment, with a discount factor of 0.1.

3.7 Conclusion

We developed Mobius, a scheduling system that can deliver both high throughput and fairness in shared mobility platforms. Mobius uses the insight that, when operating over rounds, scheduling on the convex boundary of feasible allocations, as opposed to the Pareto frontier,

provably improves on fairness with time. We showed that Mobius can handle a variety of spatial and temporal demand distributions, and that it consistently outperforms baselines that aim to maximize throughput or achieve fairness at smaller timescales. Additionally, through real-world ridesharing and aerial sensing case studies, we demonstrated that Mobius is versatile and scalable.

There are several opportunities for extending the capabilities of Mobius. First, Mobius assumes that customers are not adversarial. Developing strategyproof mechanisms that incentivize truthful reporting of tasks by customers is an open problem. Second, we design Mobius to only balance customer throughputs. We believe the optimization techniques we developed (§3.4) can be extended to support other platform objectives, such as task latency, vehicle revenue, and driver fairness. Finally, incorporating predictive scheduling, where the platform can strategically position vehicles in anticipation of future tasks, is an interesting direction for future work, as it can further increase platform throughput.

Chapter 4

Application-Level Service Assurance with 5G RAN Slicing

A rapidly growing number of mobile applications—such as mixed reality, cloud gaming, video conferencing, and cloud robotics—require predictable network connectivity (i.e., throughput and latency). The 3GPP specifications for 5G Radio Access Networks (RANs) recognized this requirement for next-generation mobile apps and introduced *network slicing* [37], a virtualization primitive that allows an operator to run multiple differentiated virtual networks, called slices, atop a single physical network. A slice can support a set of users or a set of applications with similar connectivity requirements. It can span multiple network domains, including the radio access network (RAN) [30, 48, 76], core [81, 125], transport [108] and fronthaul [23]. Operators can distribute resources, like physical resource blocks (PRBs) in the RAN, amongst the slices to provide differentiated connectivity. RAN slicing is of particular interest for service assurance [28] since the last-mile wireless link is often the bottleneck for mobile apps [9, 14].

Existing approaches [18, 30, 48, 76, 130] allocate PRBs to different slices to guarantee slice-level service assurance, e.g., through service-level agreements (SLAs) for total slice throughput. However, in order to realize the vision of network slicing, where apps achieve the network performance that they require, the service assurance should be provided at

application-level. Existing approaches fall short of enabling operators to provide this important capability. Slice-level service assurance does not guarantee throughput and latency to *each app* in the slice, since different users in the same slice can experience wildly different channel conditions, as we explain in §4.1. We need app-level service assurance in order to meet the requirements of each app within a slice. However, two key challenges arise when optimizing for app-level service assurance:

Challenge #1: Search space complexity

Prior approaches [18, 48, 76, 130] provide slice-level service assurance by tracking a state space consisting of aggregate slice-level statistics, including the average channel quality of all users in a slice, the observed slice throughput, etc. To extend these slice-level methods to support app-level requirements, one could potentially expand the state space to track the channel quality, the observed throughput, and the observed latency experienced by each app in a slice, essentially treating each app as a slice for the purposes of service assurance. However, the resulting state space, consisting of all possible values that the tracked variables can take, grows exponentially in the number of apps, rather than the number of slices. Further, the control policy involves searching through this state space to determine an allocation of PRBs to slices that complies with the SLA constraints. This results in an intractable optimization problem for practical deployments, where each slice accommodates tens to hundreds of apps (§4.1.2).

Challenge #2: Determining resource availability

To compute slice bandwidth allocations within the total available bandwidth, operators typically run admission controllers that admit or reject incoming apps according to a policy that depends on slice monetization preferences, fairness constraints, and other objectives. Algorithms for admission control have been studied widely [24, 25, 96, 114] and are not the focus of this paper. However, operators need a way to determine if the RAN has resources to accommodate the SLAs of an incoming app, in addition to the apps already admitted. We

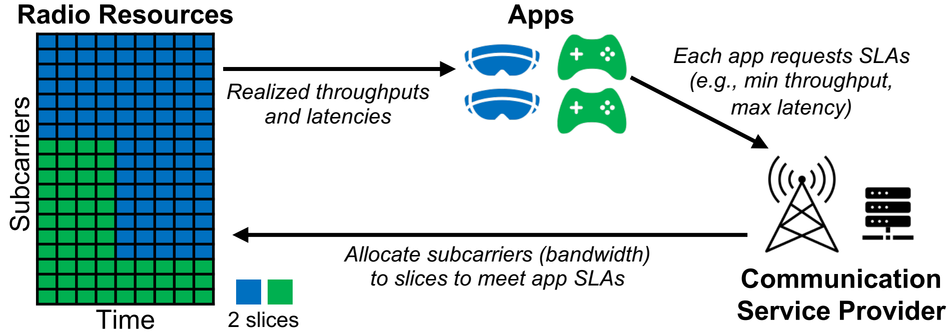


Figure 4-1: Apps express their connectivity requirements in terms of SLAs, and the operator provisions slice bandwidths to fulfill all SLAs.

cannot adapt prior approaches [56, 68, 75], which compute required PRBs to support slice-level SLAs, because the state-space complexity precludes treating each app as a slice (§4.1.3).

This paper presents *Zipper*, a real-time RAN slicing system that dynamically allocates PRBs (i.e., bandwidth) to network slices to ensure app-level throughput and latency SLAs for every app in every slice.[1] As illustrated in Fig. 4-1, under this model, apps express their network requirements to the operator in the form of SLAs, i.e., minimum throughput and maximum latency. The operator then fulfills these SLAs over the shared wireless medium by computing and allocating the PRBs required by each slice. This paradigm of operators provisioning connectivity, so that each app meets its desired network requirements, is similar to the familiar model of cloud computing—where the developer requests a combination of compute, memory, and I/O bandwidth for a particular workload, and the cloud service provider finds the right allocation of resources to reliably deliver the desired performance.

Zipper addresses the challenges in enabling app-level service assurance via three contributions:

- To manage the search space complexity, we decouple the network model and the control policy by formulating SLA-compliant PRB allocation as a **model predictive control (MPC)** problem. Zipper uses standalone predictors to forecast each of the tracked state space variables, such as the wireless channel experienced by each app. It then feeds these predictions into a control algorithm that computes a sequence of future bandwidths for each slice based

[1] We focus on app-level, but our solution also generalizes to the user-level.

on the predicted state. Our insight is that Zipper does not need to enumerate different future states within the state space, by using the well-known MPC framework (§4.2.1).

- We propose an **efficient control algorithm** to allocate PRBs (i.e., bandwidth) amongst the slices. Zipper efficiently prunes the search space of possible PRB allocations using the insight that app throughput and latency vary monotonically with the number of PRBs (§4.2.2).
- We **forecast RAN resource availability**, guided by the following question: for an incoming app A, does the RAN have enough PRBs to admit A, given the other apps already admitted? Naively applying Zipper's bandwidth estimation algorithm for a distribution of possible channel conditions experienced by the app resulted in prohibitive estimation times. We instead design a family of deep neural networks (DNNs) to predict the distribution of required PRBs. We train these neural networks on simulations of Zipper's control algorithm offline and then apply them to predict the resource availability in real time (§4.2.3).

We design an O-RAN-compatible [64] architecture to realize these algorithmic concepts (Fig. 4-4). We have implemented Zipper atop a production-class end-to-end 5G vRAN platform, implementing hooks [49] across different modules in vRAN Distributed Unit (DU) to control slice bandwidth dynamically without compromising real-time performance (§4.3). On a typical RAN workload consisting of video streaming, conferencing, IoT, and virtual reality apps, our real-time system can support up to 200 apps and over 70 slices on a 100 MHz channel. We find that Zipper outperforms prior schedulers and slicing frameworks (§4.4); relative to a slice-level service assurance scheduler [76], Zipper reduces SLA violations, measured as a ratio of the violation of the app's request, by 9×.

4.1 Problem Setup and Challenges

In this section, we formalize the optimization problem of providing app-level throughput and latency assurance, and illustrate, through a toy example, the challenges in computing

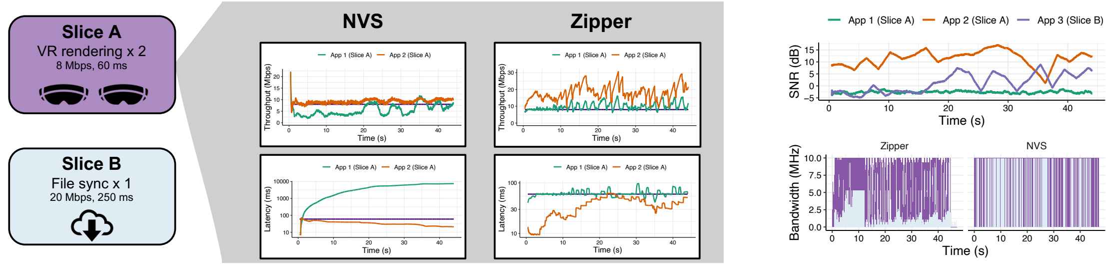


Figure 4-2: Zipper can efficiently manage an expressive and comprehensive state space to deliver SLAs for each app in each slice.

slice bandwidth allocations efficiently.

4.1.1 Problem Formulation

Zipper allocates slice bandwidths to meet app-level throughput and latency SLAs, while (i) each app's wireless channel quality fluctuates and (ii) apps join and leave the network asynchronously. We assume that the operator configures its RAN with a set of slices, catering to different traffic types (e.g., cloud gaming, video streaming, etc.) and to different enterprise policies (e.g., separate slices for Zoom and Microsoft Teams sessions). The operator configures each each slice with a particular MAC scheduler, which is responsible for allocating PRBs to apps in each slice. Fig. 4-2 illustrates a RAN serving two slices: slice A for VR remote rendering and slice B for video downloads (e.g., video editing).

Formalizing SLAs

We assume that each app selects a slice, based on its specific throughput and latency requirements.[2] For example, in Fig. 4-2, the two VR apps each require a minimum throughput of 8 Mbps and a worst-case latency of 60 ms, while the file sync app requests a minimum throughput of 20 Mbps and a worst-case latency of 250 ms. Let x_a^{SLA} and d_a^{SLA} denote the throughput and latency SLAs for an app a.

Let $\bar{x}_a(t)$ be the average throughput over a moving window of T_w slots. App a requires

[2]Alternatively, the slice controller can automatically match the app to a slice already catering to apps with similar connectivity requirements. We leave app-to-slice matching to future work.

that $\bar{x}_a(t) \geq x_a^{SLA}$. Similarly, let $\bar{d}_a(t)$ be the average latency over T_w. When app a expresses a latency SLA, it requires that the average latency $\bar{d}_a(t) \leq d_a^{SLA}$.

Formalizing slice bandwidth allocation

We formalize the optimization problem to compute SLA-compliant schedules. Since there can be multiple valid allocations that satisfy the SLAs, we choose the one that minimizes the total bandwidth:

$$\underset{\mathcal{S}_s(t), B_s(t) \forall s \in S \, \forall t}{\text{argmin}} \quad \sum_t \sum_{s \in S} B_s(t) \tag{4.1}$$

$$\text{s.t.} \quad \sum_{s \in S} B_s(t) \leq B \qquad \forall t \tag{4.2}$$

$$\bar{x}_a(t) \geq x_a^{SLA} \qquad \forall a \in A_s \, \forall s \in S \, \forall t \tag{4.3}$$

$$\bar{d}_a(t) \leq d_a^{SLA} \qquad \forall a \in A_s \, \forall s \in S \, \forall t, \tag{4.4}$$

where B is the total bandwidth available at the base station, S is the set of network slices, and $A_s(t)$ is the set of apps subscribed to slice $s \in S$ at time t. $B_s(t)$ denotes the bandwidth allocated to slice s in scheduling round t. $\mathcal{S}_s$ is the MAC schedule for slice s.

At each timestep t, Zipper must select $B_s(t)$ for each slice $s \in S$ such that the throughput and latency SLAs for all apps $a \in A_s(t)$ are satisfied, as captured by the constraints in Equation (4.3) and Equation (4.4) respectively. Equation (4.2) ensures that the sum of slice bandwidths does not exceed the bandwidth available at the base station. The objective in Equation (4.1) states that Zipper must find the sequence of slice schedules and corresponding bandwidths that minimizes the overall spectral utilization.

This approach differs from previous approaches that compute the minimum bandwidth required by each slice to satisfy slice-level SLAs, such as average slice throughput. For example, Fig. 4-2 visualizes the results achieved by NVS [76], a widely-used slice-level service assurance system [18, 36, 48], for our toy example with two slices. Notice that NVS is *not* able to meet

the throughput for App 1, and instead overcompensates for App 2. However, directly extending slice-level service assurance approaches to satisfy app-level SLAs explodes the state space.

Sometimes, the network could be at capacity, and the formulation in Equation (4.1)-Equation (4.4) will not have a valid solution. To make the problem tractable, we can relax the constrains in Equation (4.3) and Equation (4.4) into two penalty functions that quantify how far—if at all—an app deviates from its throughput and latency SLAs:

$$f_x^a(t) = \left|\min\left(\overline{x}_a(t) - x_a^{SLA}, 0\right) / x_a^{SLA}\right| \tag{4.5}$$

$$f_d^a(t) = \left|\min\left(d_a^{SLA} - \overline{d}_a(t), 0\right) / d_a^{SLA}\right| \tag{4.6}$$

$f_x^a(t)$ in Equation (4.5) is nonzero only when the throughput is less than the SLA and measures the deviation as a fraction of the SLA. Similarly, $f_d^a(t)$ measures the deviation as a fraction of agreed-upon latency SLA, if that SLA is violated. So, we modify the objective of Zipper (Equation (4.1)) to include a term that minimizes these penalties. In practice, penalties will remain to close to 0 most of the time, since operators admit/reject incoming apps by determining whether the RAN has sufficient capacity.

4.1.2 Challenge: State Space Complexity

Prior methods [76, 130] monitor aggregate state variables like average slice throughput [76], average channel quality across all users, and average latency across all users [130], to deliver service assurance at the slice level. The search space for such state vectors grows exponentially with the number of slices. Fig. 4-2 shows that considering a slice level state space could yield poor app performance. While the slice-level method meets the overall slice throughput SLAs, it violates App 1's SLA because App 1 has an inferior channel quality to that of App 2.

We could expand the state space by considering app-level characteristics, e.g., average measured app throughput, average measured app latency, channel quality of each user, etc. We could then extend slice-level service assurance approaches to meet app-level SLAs,

treating each app as an individual slice for the sake of service assurance. However, the state space grows exponentially with the number of apps, rather than with the number of slices. The number of apps served by each slice in a base station could range from tens to hundreds, resulting in an intractable state space to deliver real-time performance. For example, LACO [130] trains an agent using reinforcement learning to learn a policy that selects slice bandwidths. If we adapt that architecture to a fine-grained state space, the training complexity explodes, since the agent needs to explore a more expansive search space.

4.1.3 Challenge: Determining RAN Resource Availability

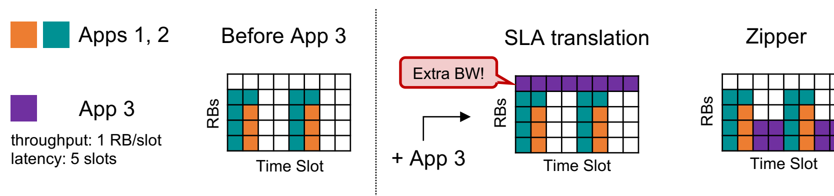


Figure 4-3: Translating an app's SLAs directly to required slice bandwidth can ignore schedules with greater spectral efficiency.

Recall that the slice controller's bandwidth allocations B_s cannot exceed the total available bandwidth B. As load at the RAN increases, it becomes more challenging to fulfill all SLAs under this bandwidth constraint. As a result, operators typically run admission controllers on top of their slicing systems, only admitting apps that can receive the requested SLAs. Admission control policies can depend on a variety of objectives, such as slice monetization preferences, operational costs, fairness, energy constraints, etc. Admission control for network slicing has received significant attention over the years [24, 25, 96, 114], and is not the focus of this paper. However, in order to interface the slice controller with a particular policy, we need a mechanism [96] that answers the following question: *for a given contract duration, can the RAN fulfill the SLAs for an incoming app without compromising on commitments made to existing apps?*

NVS adds a buffer to each slice [76] to absorb errors that operators make in admitting

apps that it cannot support. However, a constant buffer can underutilize the spectrum. Prior work [68] injects the incoming app into a separate "best effort slice" and observes whether it achieves its SLAs to determine if the RAN has resources in the desired slice to accommodate this app. However, performance in the "best effort" slice may not faithfully represent performance in the target slice. A more analytical approach [56, 75] is to translate the SLAs for the incoming app into a measure of resource blocks required to support that app in the desired slice via an analytical model or a lookup table that maps SLAs to a PRB requirement. These methods can waste spectrum.

Consider the example in Fig. 4-3, where a slice in the RAN initially serves Apps 1 and 2. We show the resource block schedule for these two apps; notice that the RAN allocates 4 RBs of bandwidth to the slice. Our goal is to determine how much bandwidth is required to accommodate App 3, who has a throughput SLA of 1 RB/slot and a latency SLA of 5 slots. Simply translating the 1 RB/slot throughput SLA for App 3 to RB overhead, would lead us to allocate an extra RB of bandwidth to the slice. Zipper, by contrast, accommodates App 3—along with Apps 1 and 2—without adding any more bandwidth.

SLAs are two-dimensional (i.e., throughput and latency), and a slice could have an arbitrarily complex PRB scheduler, whose behavior depends on additional factors, such as the status of app queues at the base station and changing MCS in response to the wireless channel. It is therefore challenging to map SLAs to a PRB differential via an analytical model.

We need a primitive to determine RAN resource availability for an incoming app that generalizes to MAC schedulers and apps with different demand patterns. Recent RAN slicing systems [30, 48] do not address how to interface their slice controllers with operators' admission control policies. Since the RAN is often the bottleneck link [9, 14], it is often oversubscribed. Thus, slicing systems are unusable in practice without a mechanism to estimate resource availability and an accompanying admission control policy.

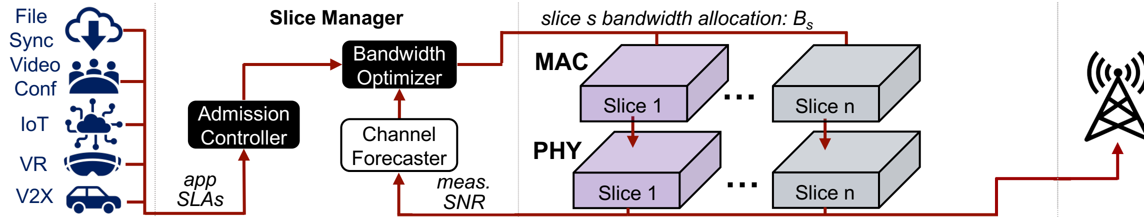


Figure 4-4: Zipper provisions connectivity by dynamically optimizing network slice bandwidth and resource allocation to meet app-level SLAs.

4.2 Design

In this section, we describe how we design Zipper, illustrated in Fig. 4-4 to enable app-level service assurance. Zipper consists of a model predictive control (MPC) framework to manage the search space complexity (§4.2.1), uses an efficient algorithm to compute slice bandwidth allocations within this MPC formulation (§4.2.2), and exposes a primitive to help operators forecast RAN resource availability (§4.2.3).

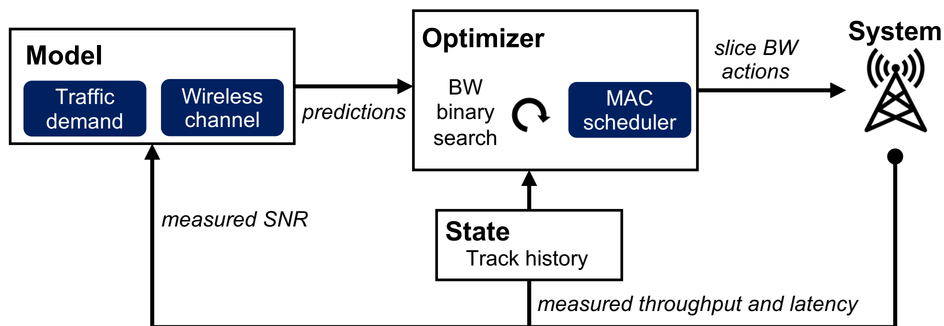
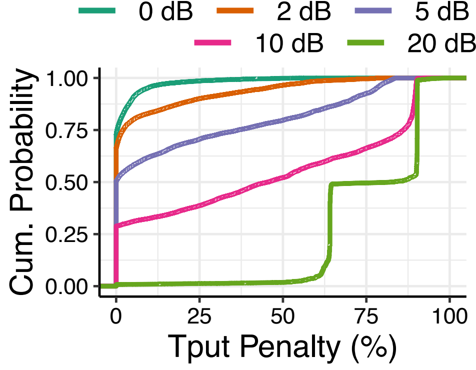


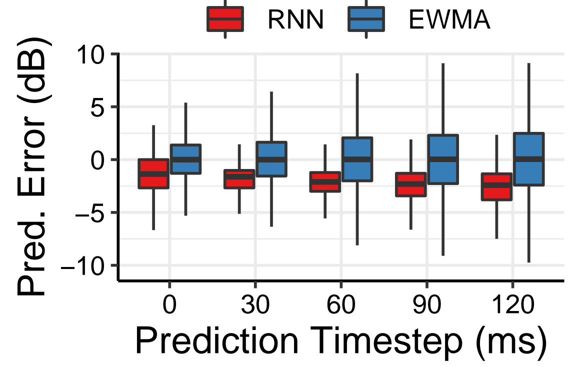
Figure 4-5: Zipper uses model predictive control (MPC) to compute slice bandwidths that comply with all app SLAs. With MPC, Zipper decouples prediction from control to manage the state space.

4.2.1 Model Predictive Control

MPC [50] is a framework to solve sequential decision making problems over a moving look-ahead horizon. It decouples a controller, which solves a classical optimization problem, from a predictor, which explicitly models uncertainty in the environment. MPC has proven



(a) Resilience to SNR error.



(b) SNR prediction error.

Figure 4-6: Zipper is resilient to modest ∼2 dB error in forecasting SNR. Its MPC framework supports different channel forecasters. While both have small median errors, the RNN model outperforms EWMA.

practical in a number of real-world control problems, including in adaptive bitrate selection for video streaming [122, 127] and in robotics [120].

Fig. 4-5 illustrates how Zipper applies MPC to solve the optimization problem formulated in Equations 4.1-4.4. The state space consists of (i) the average throughput and average latency experienced by each app over the past T_w slots, (ii) the average signal-to-noise ratio (SNR) of each user over T_w as a measure of the channel quality, and (iii) the incoming data traffic. The action space consists of a bandwidth allocation B_s for each slice s. Independent forecasters predict how each of the state space variables evolves over a short term planning horizon. The controller uses these predictions to determine the bandwidth schedule $B_s(t)$ for each slice.

MPC allows us to use network models to explicitly predict the future states over the short term, and thus avoid searching over different future states within the state space. We describe predictive models for each of the state space variables ahead.

Forecasting the wireless channel

Zipper supports different channel predictors that forecast how each app's wireless channel (i.e.,

SNR)[3] will evolve over the near term. Forecasting the wireless channel is a well-researched and fundamentally challenging problem [69, 79, 83, 91]. We acknowledge this in our design, and do not aim for a perfect predictor. Instead, we quantify a desirable performance for our bandwidth allocation task, and then propose methods that meet that target.

To understand the impact of SNR prediction error on Zipper's ability to meet SLAs, we run a simple experiment,[4] involving a 40 MHz channel and 10 video conferencing apps (i.e., 2 Mbps min throughput, 150 ms max latency), split across 2 slices. We randomly assign each user a 30-second SNR trace gathered on a production 5G network. To understand the effect of prediction error in the worst case, we introduce a dummy predictor that simply returns the ground truth SNR value added to some constant prediction error. Fig. 4-6a visualizes the results as CDFs of the throughput and latency penalties ($f_x^a(t)$ and $f_x^a(t)$ in Equation (4.5) and Equation (4.6) respectively) for different prediction errors. As expected, larger prediction errors lead to higher penalties. However, notice that a small (but not insignificant) error of 2 dB has a modest impact on penalty.

We also observe that the MAC scheduler uses the SNR forecast to determine what MCS to assign an app. The 3GPP standards define 32 MCS values [66], and MAC schedulers use a lookup table to map measured channel quality to MCS [72]. We find that this table is quantized in 2 dB steps, so a prediction error of under 2 dB may still yield the correct MCS.

To forecast each user's channel, we train a sequence-to-sequence Recurrent Neural Network (RNN) [105], which uses an input sequence of SNR measurements over the last 1 second to predict a sequence of SNR measurements over the next 150 milliseconds. App. B.1.1 describes the architecture of this RNN. We train this model using a dataset of SNR traces [101] collected over a commercial network at scale. We evaluate the accuracy of this RNN on a holdout set from the same bank of traces. Fig. 4-6b shows the prediction error (relative to the ground truth) at different points over the 150 ms prediction horizon. Each boxplot

[3] For simplicity, we forecast SNR as an aggregate quantity over all subcarriers in a given time slot. However, Zipper can support richer predictors and schedulers [30] predict SNR at a subcarrier granularity.

[4] We evaluate Zipper more extensively against an Oracle policy in §3.6.

shows a distribution of error over all traces in our holdout set. We compare the accuracy of our RNN against a simpler predictor that tracks the SNR with an exponentially-weighted moving average (EWMA). Both predictors have a suitable median performance, which falls within our target of ∼2 dB error. However, the RNN has a more consistently tight distribution. Moreover, as we describe below, prediction errors at later timesteps are less consequential, since Zipper recomputes fresh allocations at a fine granularity.

Although we use this RNN model in our implementation, it is not a contribution; Zipper could use any other predictor—including the EWMA filter—with comparable error.

Other state space variables

Zipper tracks the average throughput and latency, $\overline{x}_a(t)$ and $\overline{d}_a(t)$ respectively, experienced by each app in the past T_w time slots. Since Zipper only tracks historical averages, there is no need for prediction. We assume that the traffic demand for each app follows the throughput SLA requested by the app. If the traffic demand from an app is higher than the agreed upon throughput SLA, Zipper only ensures that it fulfills the negotiated SLAs. Tuning slice bandwidths using more detailed traffic demand predictions is part of future work.

4.2.2 Tuning Slice Bandwidths Efficiently

Given the state space, defined as each app's SNR, average throughput, average latency, and traffic demand, the slice manager must find the most spectrally-efficient slice bandwidth allocation that satisfies the SLAs, as we formalize in §4.1.1.

One approach to is to analytically derive a function that maps SLAs to a valid slice bandwidth. However, this is challenging, since the expected throughput and latency of any given app depends not only that app's channel quality and queue status at the base station, but also on the characteristics of the other apps contending for the same radio resources in the slice.

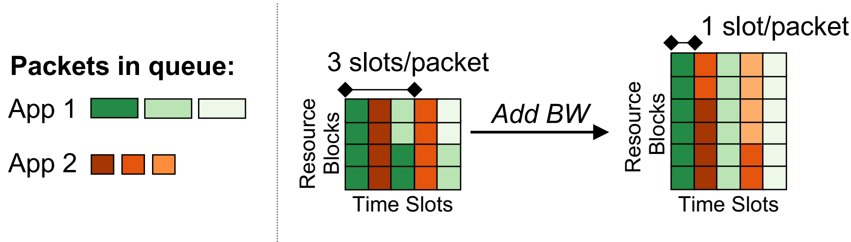


Figure 4-7: Exposing more bandwidth to a slice reduces packet latency.

Monotonicity

Our insight is that both app throughput and app latency are *monotonic functions of slice bandwidth.* Fig. 4-7 illustrates this monotonicity property for latency with an example. Consider two apps (green and orange) with different packet sizes, and a round-robin MAC scheduler. We define latency as the difference between the times at which (i) the first byte of a packet arrives at the base station and (ii) the last byte of a packet is sent over the air. The diagram on the left visualizes a round-robin schedule for a slice with 4 resource blocks. Notice that the green app's packet is spread across multiple slots, and since the MAC is round-robin, the packet's latency is at least 3 slots. By contrast, when the bandwidth is 6 resource blocks (diagram on the right), the packet latency is just 1 slot. Thus, per-packet latency is a monotonically-decreasing function of slice bandwidth. Similarly, the app throughput increases monotonically with slice bandwidth. App. B.1.2 elaborates on this property.

Because app throughput and latency vary monotonically with slice bandwidth, there exists a minimum bandwidth B_s for $s \in S$ that satisfies all SLAs. Therefore, a solution that minimizes Equation (4.1) is one that minimizes the individual slice bandwidths, and Zipper can optimize each slice independently.

Computing bandwidths

Zipper treats slice MAC schedulers as a blackbox; the search algorithm does not need to know the scheduling logic. Instead Zipper uses each slice's scheduler as a building block to find the smallest bandwidth that satisfies the SLAs of all apps in the slice. In each time

interval t, Zipper computes $B_s(t)$ using an iterative algorithm that *simulates* the MAC scheduler for different candidate bandwidths $\tilde{B}_s$. Zipper queries the MAC scheduler for each $\tilde{B}_s$. Zipper supplies the MAC scheduler with (i) all outstanding packets in each app's queue, and (ii) a forecast of each app's channel quality over the scheduling horizon (§4.2.1). Zipper evaluates the resulting schedule $\tilde{\mathcal{S}}_s$ by computing the *penalty scores*, $f_x^a(t)$ and $f_d^a(t)$ as defined in Equation (4.5) and Equation (4.6), respectively, to determine if the schedule satisfies the SLAs. Then, amongst the set of valid schedules (i.e., when $f_x^a(t) = f_d^a(t) = 0$), Zipper chooses the one that requires the least slice bandwidth, i.e., the smallest $\tilde{B}_s$.

Zipper navigates the search space of candidate bandwidths using binary search. It starts with the entire range $0 \leq \tilde{B}_s \leq B$, and prunes the search space by half in each iteration. In the first iteration, Zipper computes a MAC schedule $\tilde{\mathcal{S}}$ for the allocation $B/2$. For instance, if at least one app's throughput in $\tilde{\mathcal{S}}$ is less than the throughput agreed to in the SLA, then Zipper determines that the slice needs more bandwidth to satisfy the constraint; so it continues the search in the range $(B/2, B]$. By contrast, if all performance metrics comply with the SLA constraints, then Zipper determines that the slice could possibly meet the SLAs with less bandwidth; so it continues the search in the range $[0, B/2)$. We can apply this binary search optimization because app throughput and latency vary monontonically with slice bandwidth.

Zipper computes schedules in a cascading manner, where, in each timestep t, Zipper solves the MPC problem for a finite future horizon of T_h. It recomputes its allocation every $T_e \leq T_h$ in order to incorporate recent snapshots of user channel and app queue statuses. We use $T_h = 150$ ms and $T_e = 50$ ms[5] to ensure that Zipper is reactive but not myopic. If $B_s(t)$ violates Equation (4.2) for any slice s, Zipper resolves the conflict to ensure that the allocated bandwidth does not exceed the capacity.

[5]We found that Zipper was not very sensitive to T_h and T_e; we selected $T_e = 50$ ms because our SNR predictors are most accurate over this horizon (§4.2.1). A shorter horizon allows us to replan with fresh estimates of SNR.

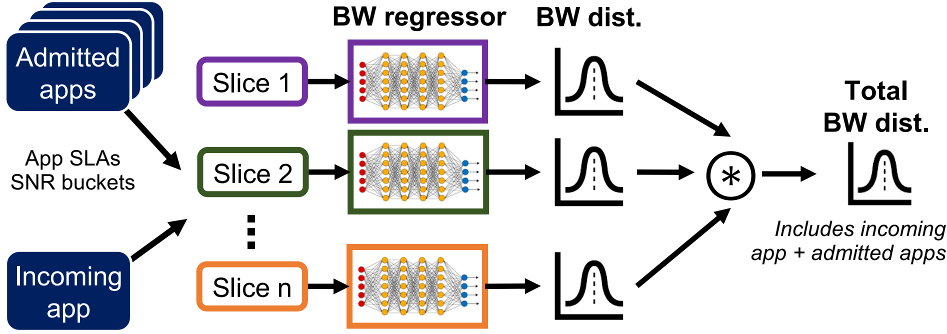


Figure 4-8: Zipper builds a family of DNNs that forecasts bandwidth distributions for slices consisting of different MAC schedulers and apps with different demand patterns.

Resolving conflicts

Since Zipper tunes slice bandwidths independently, the allocations could conflict, i.e., $\sum_{s \in S} B_s(t) > B$, which violates Equation (4.2). To resolve conflicts, Zipper deducts—from each slice—the excess bandwidth in proportion to the share of bandwidth that each slice was originally allocated. In practice, Zipper will not trigger this step often because it gates incoming requests with an admission controller (§4.2.3).

App. B.1.3 provides pseudocode for how Zipper computes slice bandwidth allocations. Note that Zipper only considers the number of resource blocks in aggregate when allocating resource blocks to a slice. In future work, we can extend Zipper to determine the most suitable set of resource blocks given the channel conditions, using the techniques proposed by RadioSaber [30]. We can also model methods to increase user capacity, such as beam steering [26, 52].

4.2.3 Forecasting RAN Resource Availability

To estimate if the RAN has resources to support an incoming app, Zipper answers the following question: for the contract duration, does the RAN have enough PRBs to accommodate the incoming app and to fulfill SLAs for all other admitted apps?

Predicting bandwidth statistics

Zipper estimates the distribution of bandwidths, i.e., number of PRBs, that each slice will require over a predetermined contract duration—including the incoming app. Translating the SLAs of each app in a slice to radio resource requirements [56, 75] can yield overestimates of the required bandwidth. Instead, Zipper simulates its slice manager over thousands of channel traces. Direct simulations capture how the slice MAC exploits statistical multiplexing to fulfill the SLAs without adding bandwidth to a slice (§4.1.3). However, running thousands of simulations for a reasonable contract duration (e.g., 5 mins) is expensive, since the slice manager computes and evaluates many MAC schedules.

All we need from the simulations is the bandwidth statistics—not the PRB schedules. To approximate the bandwidth statistics, Zipper develops a family of deep neural networks (DNNs), instead of running thousands of micro-simulations at runtime. Fig. 4-8 illustrates the design of this module. Since each slice caters to apps with similar network requirements (i.e., SLAs), we tailor a DNN for the traffic characteristics of each slice, similar to lookup tables in prior work [56, 75] that translate SLAs to PRB requirements. Each DNN treats a slice's MAC scheduler as a blackbox process and learns the nonlinear relationship between inputs—app demand patterns, SLAs, and channel quality—and the required bandwidth.

To create a simple and tractable input embedding for the DNN, we make a few assumptions. First, we assume that all apps in a slice have the same SLAs; this is reasonable because, in practice, network slices often isolate similar kinds of traffic [48]. Moreover, slight variations in SLAs (e.g., 4 Mbps vs. 5 Mbps video conference flows) should have negligible impact on bandwidth requirements. Second, in order to discretize the space of possible SNR values, we assign each app to an SNR bucket from the set {`poor`, `bad`, `good`, `great`}, where each bucket corresponds to a range of SNR values (e.g., -5 dB $\leq$ `bad` $<$ 2 dB).[6] Zipper drops each incoming app into the best effort slice for a brief period (e.g., 5 seconds),

[6] Note that we only discretize SNR into bins to estimate resource availability with the DNNs; at runtime, the slice controller forecasts SNR (§4.2.1).

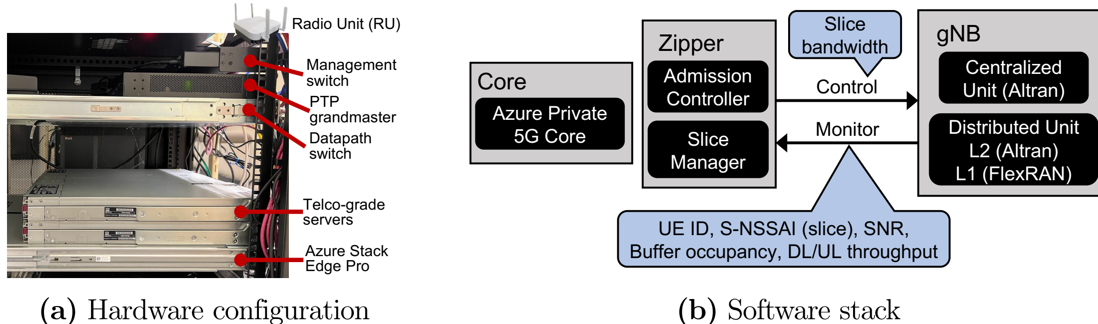
computes its median SNR, and assigns it an SNR bucket. For existing apps, Zipper uses SNR measured over the lifetime of each app to determine the most suitable bucket. Note that the best effort slice is only for measuring SNR, not for assessing resource availability. For each slice, Zipper generates a feature embedding consisting of the number of apps in the slice (including the incoming app, if applicable), and the number of apps in each SNR bucket. App. B.2.1 describes the DNN architecture.

Training DNNs

We generate training data by using Zipper's slice manager as a simulator. We start by enumerating all possible feature embeddings, and run a micro-simulation of Zipper for each embedding using simulated channel traces (assuming a Rayleigh channel model) with SNR values corresponding to the SNR bucket for that embedding. We prune the space of embeddings using some simple heuristics. For instance, if we find that a simulation of 55 apps—all with `poor` SNRs—requires a maximum bandwidth of 100 MHz, we discard all embeddings with 56 or more `poor` apps, since those configurations will also require at least 100 MHz.

Estimating resource availability

Each DNN returns slice bandwidths as a probability distribution P_s for slice s. To compute a distribution of the required spectrum, we convolve[7] the slices' independent probability distributions: $P = P_1 \circledast P_2 \circledast \cdots \circledast P_n$. P is the forecasted distribution of required bandwidth. Operators can choose a suitable percentile of P (e.g., p95, p99, etc.) based on their tolerance preferences. For instance, a conservative policy might deem resources available for the incoming app if the p99 bandwidth is less than the total available bandwidth B.



(a) Hardware configuration

(b) Software stack

Figure 4-9: We implement Zipper atop a production-class 5G network.

4.3 Implementation

We implemented Zipper atop an end-to-end production-class 5G network built on virtual RAN (vRAN) and Core components. Our testbed uses commercial off-the-shelf user devices.

Testbed hardware

As illustrated in Fig. 4-9a, our production testbed consists of two 32-core HPE ProLiant DL110 Gen10 Plus Telco-Grade servers with Intel Xeon-Gold 6338N CPUs and an Azure Stack Edge Pro device. The servers perform all baseband processing in software, except for LDPC decoding which is performed by a lookaside accelerator—Intel eASIC ACC100—to meet the stringent real-time requirements posed by baseband processing workloads. We use Foxconn RPQN-7800 5G Open Radio Units (RUs) operating on 100 MHz channels in the n78 band. We obtained FCC STA licenses to operate the radios for experimental purposes. The radio units support the popular O-RAN split Option 7.2x [95], designed to reduce the optical bandwidth required for fronthaul traffic while keeping the RU simple and inexpensive. The radio has four antennas and supports up to four spatial streams. The RU and the Telco servers are synchronized using a Qulsar Qg2 carrier-grade PTP Grandmaster [100]. The testbed additionally includes a high-speed datapath switch Arista 7050 to carry the fronthaul traffic, as well as a management switch from Netgear to facilitate remote management of

[7]The probability distribution of a sum of independent random variables is the convolution of their individual distributions [61].

the hardware devices.

Testbed software

Fig. 4-9b illustrates the software stack. For the L1, our testbed runs production-ready Intel FlexRAN v20.11 [35]. For the L2/L3, we use Altran 5G vRAN software from Capgemini [5]. A single Telco server hosts both the Altran and FlexRAN software, while Zipper runs on another server. We installed Azure Private 5G Core (AP5GC) [87] on the Azure Stack Edge Pro device to provide 5G core services. Both Altran vRAN and AP5GC core support standard-compliant slicing and can provide differentiated service to commercial devices. We integrated all of these systems end-to-end to realize a production-class 5G network using virtualized RAN and Core components.

Zipper can programmatically control the RAN using this virtualized setup . We implemented a vRAN data collection system to retrieve SNR of each user and buffer occupancy from the 5G vRAN software. We monitor the user throughput using diagnostic information from AP5GC. We utilize the buffer occupancy and user throughput information to estimate the latency experienced by each dataflow. We use a custom slice bandwidth controller from Altran 5G vRAN that changes the slice bandwidth according to the output from Algorithm 2. The controller is lightweight, and this API allows us to adjust slice bandwidth allocations at the granularity of a few milliseconds.

Zipper is compatible with O-RAN specifications [64]. Zipper would be hosted in the Near-Real-Time RAN Intelligent Controller (RIC). In an O-RAN deployment, Zipper would retrieve SNR, buffer occupancy and throughput from the E2 Monitor interface, and would send slice bandwidth control signals over the E2 Control interface. Since app admission is not a real-time decision, we would implement the admission controller within the Non-Real Time RIC.

Zipper slice manager

Our implementation of the slice manager (Fig. 4-4), which includes the MPC framework, bandwidth allocation algorithm, and resource availability module, is about 4,000 lines of Go. The slice manager is multi-threaded. It computes the bandwidth allocations for each slice in parallel. When Zipper receives a new app request, the slice manager spawns a new thread to run the admission controller. Zipper obtains RAN telemetry (i.e., SNR measurements, app buffer occupancy [78], etc.) from the vRAN via a UDP socket. Zipper populates its state—maintained over a moving horizon T_w (§4.1.1)—with these measurements. We implemented data buffers to support quick, thread-safe read/write access that meets the stringent slot deadlines for 5G workloads [2].

4.4 Evaluation

We evaluate Zipper with typical RAN workloads (§4.4.1) on our production-grade testbed (§4.4.2) and in emulated environments (§4.4.3-§4.4.5). Our evaluation highlights include:

- On our end-to-end testbed, Zipper dynamically tunes slice bandwidths every millisecond to fulfill app SLAs (§4.4.2).
- Compared to a slice-level service assurance scheduler, Zipper reduces tail throughput and latency penalties as a percentage of app SLAs by 9× (§4.4.3).
- Zipper can support 150 apps drawn from a typical workload, and incur nearly no penalty in throughput and latency (§4.4.3).
- In order to fulfill app SLAs, Zipper utilizes about 30% more bandwidth than RAN schedulers without SLA constraints, but 50% less bandwidth than prior slicing systems (§4.4.3) .
- Zipper's admission control framework can intelligently allocate unutilized bandwidth, admitting 15% more apps for a first-come-first-served policy (§4.4.4).
- Zipper supports 200 apps and 70 slices in real time (§4.4.5).

App Type	Min Tput	Max Latency	QCI [67]	Freq.
Video conferencing	2 Mbps	150 ms	40	30%
Voice	200 kbps	100 ms	20	30%
Vehicle-to-X	200 kbps	50 ms	40	10%
Video streaming	2 Mbps	300 ms	60	20%
VR offload	10 Mbps	30 ms	68	5%
File sync	20 Mbps	—	80	5%

Table 4-9: Apps, SLAs, and frequencies selected for experiments.

4.4.1 Evaluation Setup

Emulation

We develop a real-time emulation framework to compare Zipper against baselines under controlled network environments. We develop a data generator to emulate realistic demand patterns for different apps, and develop a channel emulator that exposes apps to real network traces. We describe both below.

Apps

For our experiments, we choose several representative applications that cover the gamut of throughput and latency requirements. Table 4-9 summarizes the SLAs we select for these applications, based on the definitions in the 3GPP specifications [67], and also reports the frequency of each app type (as a percentage) in our experiments. Since we do not have access to real-world cellular traces, we mimic a typical workload according to breakdowns of mobile Internet traffic published in industry technical reports [45, 110].

For file sync apps, we instrument iPerf [43] to send UDP traffic at different rates. For video conferencing, video streaming, IoT, and v2x apps, we implement a data generator in Go to send UDP packets at different sending rates and inter-packet delays. For VR remote rendering, we gather traces from a real Hololens app, and replay the packet captures over UDP.

SNR traces

To evaluate Zipper against the baselines under a controlled setting, we use a publicly available dataset of SNR traces, collected by running mobile traffic (e.g., Netflix videos, Amazon browsing, etc.) over a production 5G network in Ireland [101]. Our testbed experiment with real client devices (§4.4.2) evaluates Zipper on a live wireless channel.

Base station configuration

For all of our experiments, we configure our base station to have a total bandwidth of 100 MHz and 4×4 MIMO (i.e., 4 layers). For simplicity, we configure all slices to numerology $\mu = 1$ (i.e., 30 kHz subcarrier spacing). In our experiments, each slice caters to apps of the same type.[8]

4.4.2 End-to-end Evaluation

We begin by evaluating Zipper end-to-end on our production-grade 5G vRAN testbed (§4.3), in order to demonstrate that Zipper can deliver reliable connectivity by dynamically adjusting slice bandwidths in real-time, while adapting to variations in channel. For this experiment, we consider a scenario where the base station has one slice that serves a single file sync app. We would like to see that Zipper (i) allocates minimal slice bandwidth such that it does not always use all 100 MHz available, and (ii) adapts the bandwidth allocation for this slice as the measured channel quality varies. We run a 17 Mbps iPerf flow on a OnePlus mobile phone that is connected to the 5G base station running Zipper. During the download, we both walk around the room and stand in a fixed location to capture a variety of channel conditions.

Fig. 4-10 shows a stacked time series chart of the bandwidth that Zipper allocates to the slice (bottom), the application throughput (middle), and the SNR of the OnePlus phone (top). The segment highlighted in yellow corresponds to the segment of time during which the UE was stationary. Notice that Zipper reliably meets the target throughput of

[8] If the operator does not know the app type ahead of time, she could match apps with similar connectivity requirements, by clustering based on SLAs (e.g., high bandwidth only, or high bandwidth and low latency).

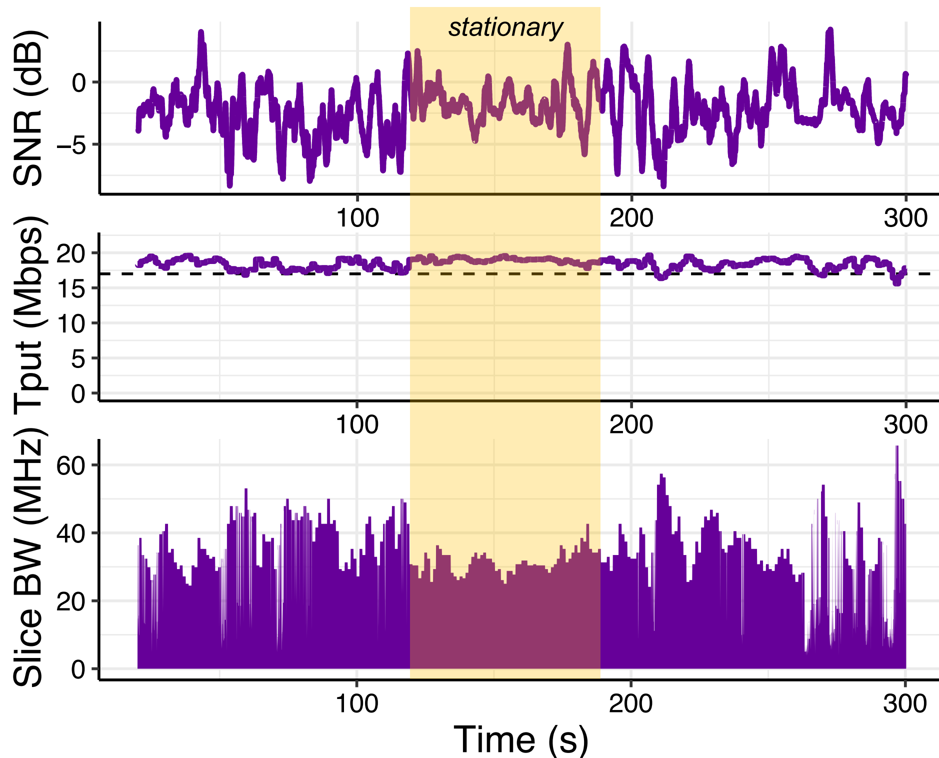


Figure 4-10: Zipper tunes the bandwidth allocated to a slice serving a mobile OnePlus phone running 17 Mbps iPerf flow.

17 Mbps—without significantly over-delivering. To do this, it adjusts its slice bandwidth allocation at a millisecond granularity; notice, in particular during the stationary period, where the measured SNR is relatively high and stable, the allocated bandwidth is accordingly lower (i.e., 30 MHz).

At the time of writing, the slice bandwidth controller from Altran [5] only supports one slice. The focus of this paper is to build a standards-compliant slicing system for app-level service assurance, without re-implementing the physical (L1) and MAC (L2) layers.

4.4.3 SLA Compliance

Setup

In order to evaluate Zipper's ability to comply with SLAs, we use our emulation framework (§3.6.1) to compare Zipper against other schemes in settings where the wireless conditions

are controlled. Like the testbed, our emulator runs in real time. We compare Zipper against the following four baseline algorithms:

- The *Single Slice policy* schedules all apps together in one slice, using a proportional fair scheduler [115], which is widely used by base stations today [9].
- The *QoS policy* [27, 129], like Single Slice, schedules all apps in one slice with a proportional fair scheduler, but additionally prioritizes each app according to its QoS Class Identifier (QCI). The 3GPP standards specify a unique QCI priority for each traffic type [67]. Table 4-9 shows the QCIs we use for the apps in our experiments. This policy is also common in production RAN deployments today.
- The *NVS policy* [76] is a dynamic network slicing algorithm for WiMAX, which provides slice-level QoS guarantees by multiplexing slices over time. Each slice requests an aggregate throughput (for all users). The NVS controller tracks each slice's throughput. In each time interval (e.g., 10 ms), it computes—for each slice—a priority, which is defined as the ratio of requested throughput to average throughput, and then selects the slice with the highest priority. While the NVS paper assumes constant MCS, we implement a modified version of the algorithm that uses channel forecasts to dynamically adjust MCS. NVS is a popular benchmark among recent RAN slicing proposals [30, 36, 48].
- The *Oracle policy* simply runs Zipper's bandwidth allocation algorithm (§4.2.2), but instead of forecasting wireless channel, it reads the true channel quality that each user will experience from the trace of SNR values. This algorithm allows us to validate our hypothesis that Zipper should be robust to modest SNR prediction error (§4.2.1).

Zipper and all baselines have access to the same overall bandwidth (e.g., 100 MHz channel). These algorithms differ in how they divide up the bandwidth amongst slices.

We vary the number of apps, ensuring, for all baselines, that apps connect in the same order and that each has the same SNR trace. We create 6 slices—one for each traffic type—and run each experiment for 3 minutes. We measure the throughput and latency penalties (Equation (4.5) - Equation (4.6)) for each app, after all apps have connected to the

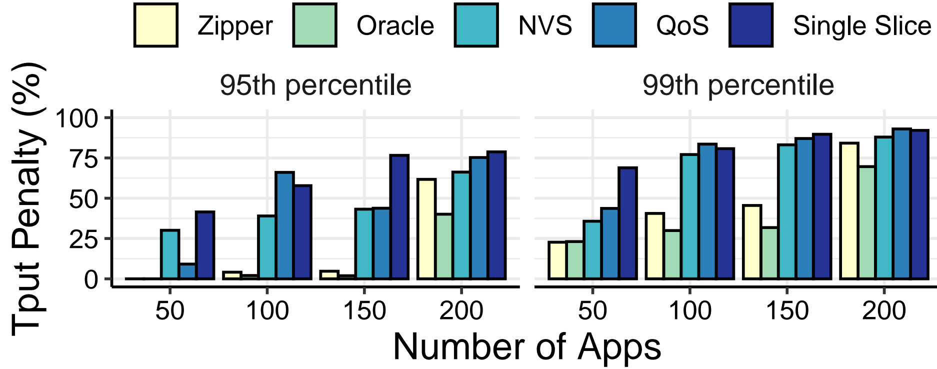


Figure 4-11: Tail throughput penalties for varying load. Apps scheduled by Zipper experience 95th percentile penalties close to 0%.

RAN (i.e., we exclude the time when apps join/leave the RAN). To create some background load at the base station, we assign twenty 20 Mbps iPerf flows to a "best effort" slice.[9] We do not use the resource availability estimator to admit/reject apps for these experiments.

Metrics of merit

We compare how well the different schemes satisfy *each app's* throughput and latency requirements, since these two quantities impact the quality-of-experience for most typical mobile apps [45, 67, 110]. In particular, for each experiment, we compute the throughput and latency penalties, defined in Equation (4.5) - Equation (4.6), and report the p95 and p99 penalties to quantify the tail performance. A lower penalty is better.

Note that we evaluate penalties (as a fraction of the requested SLAs) instead of evaluating absolute throughputs and latencies. The penalty metric allows us to directly compare app-level service assurance across slices serving apps with significantly different network requirements. Consider a slice serving a 20 Mbps file sync app A and a different slice serving a 2 Mbps video conferencing session B. A scheme that delivers 19 Mbps to A and 1 Mbps to B would yield 1 Mbps less than the requested amount for each app. But 1 Mbps is more consequential to B (50%) than it is to A (5%). The penalty metric captures this.

[9]We assign these flows the lowest QCI priority amongst those in Table 4-9.

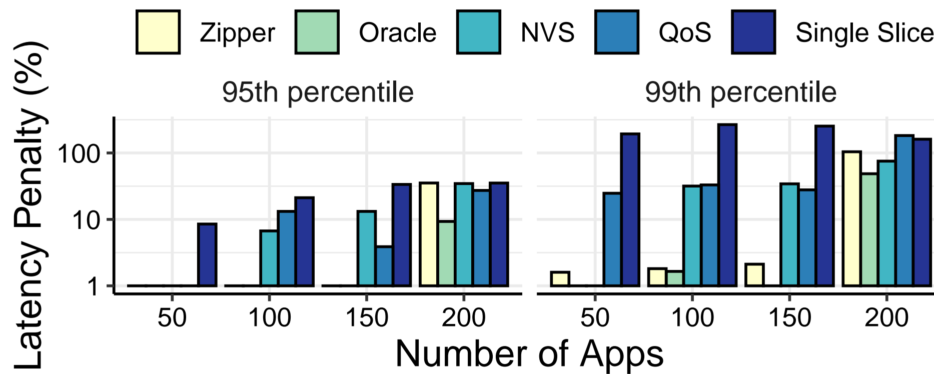


Figure 4-12: Tail latency penalties for varying base station loads. Apps scheduled by Zipper have low 95th and 99th percentile penalties.

Throughput penalties

Fig. 4-11 shows the results. The Single Slice scheduler consistently incurs the highest penalty. It cannot differentiate between traffic types, and its proportional fair scheduler will attempt to maximize throughput subject to some fairness constraints. Therefore, it tends to favor bandwidth intensive apps (e.g., file sync). Notice that, because Single Slice runs a proportional fair scheduler, it does not starve any app (throughput penalty is never 100%). The QoS policy does marginally better than Single Slice. The QCI priorities only control the relative frequencies at which the scheduler allocates resource blocks to apps, but if there is a contention, an app with higher priority may not get its desired rate. Moreover, the standards pre-define the QCIs [67], and the scheduling algorithm has no ability to dynamically tune these priorities to have the desired effect on app throughput.

NVS sees a marked improvement in penalty, compared to both Single Slice and QoS. However, the throughput penalties are still high, even at low loads (e.g., p95 penalty for 50 apps is 25%). This is because it optimizes for slice-level throughput instead of for app-level. Zipper, by contrast, exhibits comparatively lower penalties at both p95 and p99.

Latency penalties

Fig. 4-12 shows the latency penalties (on a log-scale) for the same experiment. The Single

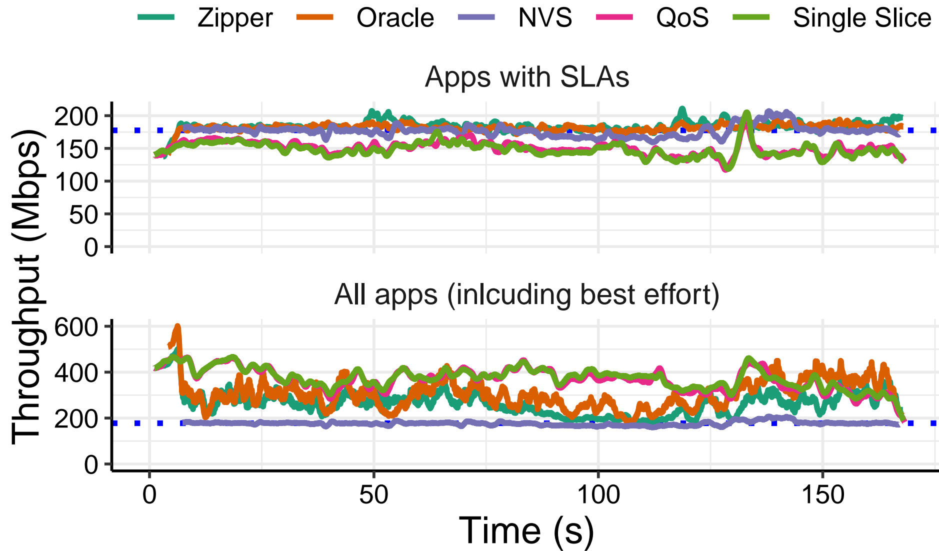


Figure 4-13: Throughput for 75 apps + 20 best effort apps. Zipper meets the SLAs reliably, and allocates excess capacity to best effort.

Slice policy delivers the worst latency penalties: it tends to favor bandwidth-intensive apps, and thus, low bandwidth, latency-critical apps (e.g., v2x and voice) will suffer. These apps, in particular, experiences latencies as high as 200 ms. QoS achieves around 40% lower penalties than Single Slice at p95, since the QCI priorities allow the scheduler to explicitly prioritize the latency-critical apps with higher QCI by scheduling them more frequently. This is because NVS multiplexes slices over time, and, in each timestep, it gives all bandwidth to the slice it chooses. Even if the switching interval is low, apps can go unscheduled for long periods of times in configurations with many slices. Zipper, by contrast, maintains very low latencies up to 150 apps, after which the base station starts to become oversubscribed.

Notice that Zipper's performance is comparable to that of the Oracle. The difference in the p95 penalty is $<5\%$ for fewer than 150 apps, which is consistent with our observations when modeling the system (§4.2.1). The gap widens slightly at 200 apps, since the system is more congested, and thus, assigning a suboptimal MCS would be more consequential. We could incorporate more sophisticated channel predictors with Zipper in the future.

RAN utilization

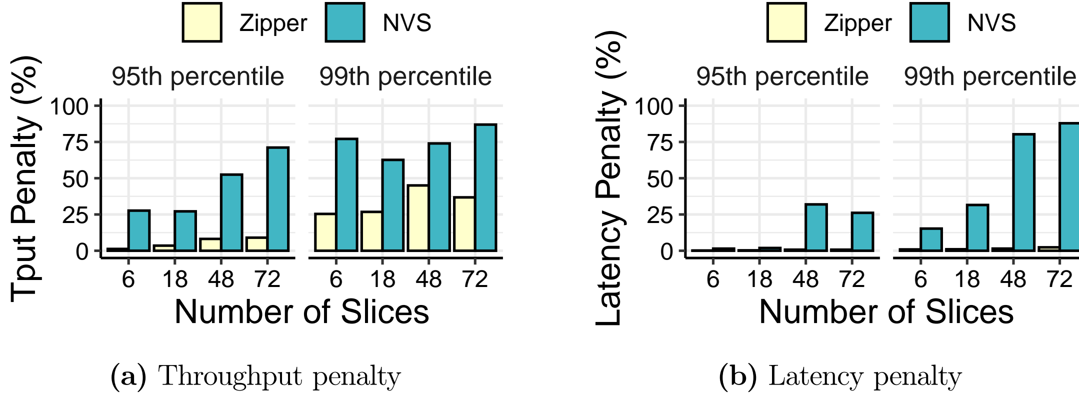
Fig. 4-11 and Fig. 4-12 show that Zipper reliably delivers the connectivity requested by each app. However, fulfilling SLAs for each app rather than for a slice in aggregate requires more bandwidth. We conduct an experiment to measure how much spectrum or capacity Zipper wastes at the expense of allocating resources to meet SLAs. Fig. 4-13 shows a 150-second time series snapshot of the aggregate RAN throughput achieved by Zipper and the different baselines for 75 apps (+ 20 best effort apps). On top chart, we show the throughput amongst the 75 apps that requested SLAs, and on the bottom chart, we plot the total RAN throughput (including best effort). The dotted blue line shows the total throughput requested by the 75 apps.

Zipper, Oracle, and NVS closely track the requested throughput at the level of slice—providing reliable and consistent performance despite the wireless channels that each app experiences. Single Slice and QoS fall about 18% below the target throughput. However, as Fig. 4-11 shows, this drop translates to far worse in terms of throughput penalty.

When we include the best effort apps, we find that QoS and Single Slice indeed achieve the highest total RAN throughput. NVS does not schedule the best effort apps because there is no excess bandwidth when all bandwidth is allocated to a single slice in a given scheduling interval. Zipper strikes a nice balance between these extremes: in addition to meeting requested SLAs, Zipper utilizes spectrum about 50% better than NVS and 30% worse than Single Slice or QoS.

Scaling up slices

The experiments so far considered 6 slices—one for each app type. However, an operator may choose to have multiple slices for the same app type, for e.g., if Zoom and Teams want to isolate their traffic. Therefore, we would like Zipper to be invariant to the number of slices. We run an experiment similar to the setup described above; we fix the number of apps at 100 and vary the number of slices. We still dedicate each slice to serving a unique app type, and we randomly assign apps to slices when there are multiple slices of the same type. Fig. 4-14a



(a) Throughput penalty

(b) Latency penalty

Figure 4-14: Zipper's performance is invariant to the number of slices.

and Fig. 4-14b show the throughput and latency penalties, respectively, for Zipper and NVS.

NVS scales poorly with the number of slices. Since NVS does not multiplex slices across frequency, slices get scheduled less frequently. Both penalties suffer with more slices, even when the switching interval is short (i.e., 10 ms). By contrast, Zipper's performance is invariant to the number of slices, since it multiplexes slices across frequency and time.

4.4.4 Forecasting RAN Resource Availability

Setup

To evaluate this module, we consider a simple first-come-first-served (FCFS) policy that admits incoming apps in the order that they arrive, as long as there is enough capacity to accommodate them without violating SLAs for other apps. Specifically, from the distribution P (§4.2.3), we admit an incoming app to its designated slice if the p95 bandwidth is within an $\epsilon = 5$ MHz tolerance of the total bandwidth B, and assigns it to the best effort slice otherwise (i.e., $P < B + \epsilon$). Note that the specific policy does not matter for this evaluation; we only want the policy to be consistent across different resource availability modules that we compare.

We compare against a "conditional" resource availability primitive: it (i) admits the incoming app into a best effort slice, (ii) measures its throughput and latency penalties for 10 seconds, and (iii) then admits in FCFS order if it incurs zero penalty or leaves it in

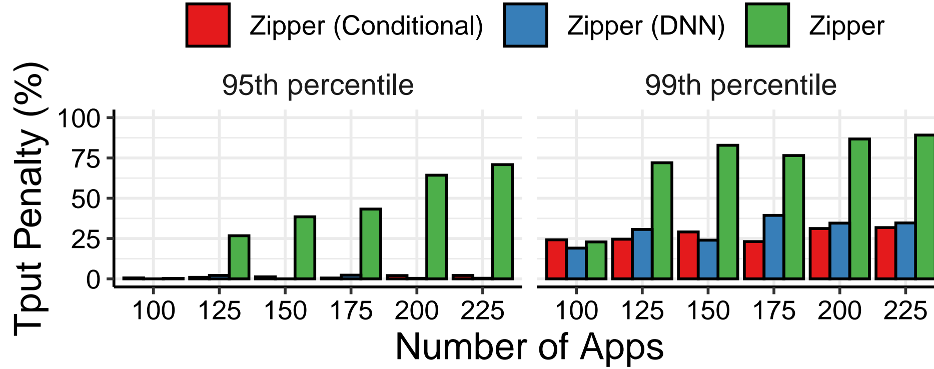
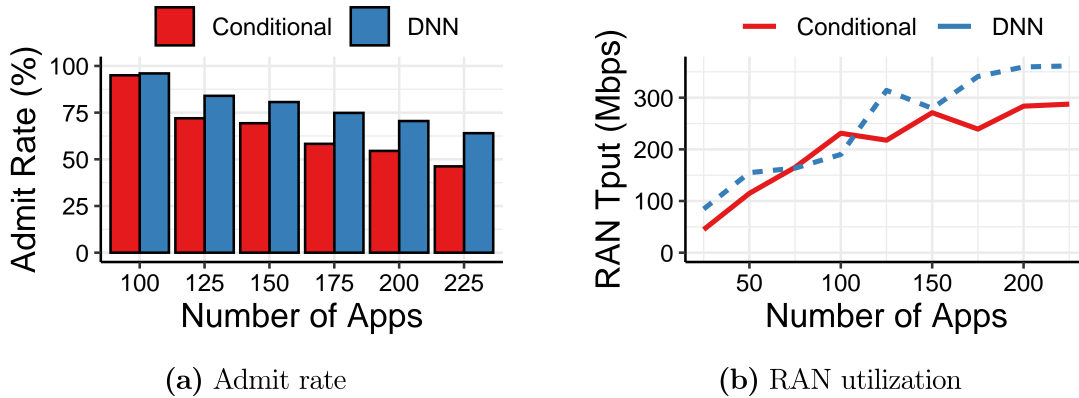


Figure 4-15: Both the conditional and DNN-based resource estimation methods achieve bounded (and low) penalties.

the best effort slice otherwise. This form of probing resource availability conditionally is common in many admission control proposals [56, 68, 96]. Our goal is to evaluate if the resource availability forecasts provided by Zipper's DNN family yield better RAN utilization than this "conditional" primitive. Both use Zipper to compute slice bandwidth schedules in real-time. As in §4.4.3, we draw apps from the distribution in Table 4-9, and select app arrival times and contract durations at random.

Bounded penalty

A good forecaster of resource availability should ensure that it can satisfy the SLAs of apps that it has already admitted before committing to a new app. This amounts to ensuring that the RAN maintains a bounded and low penalty, as more apps connect to the system. Fig. 4-15 shows the 95th and 99th percentile throughput penalties for different numbers of apps that try to connect to the RAN; the algorithm named "Zipper" uses no admission controller. Notice that, by letting each incoming app experience the network, both methods that use an admission control policy keep the penalties bounded, and, more importantly, close to 0% at the 95th percentile.



(a) Admit rate

(b) RAN utilization

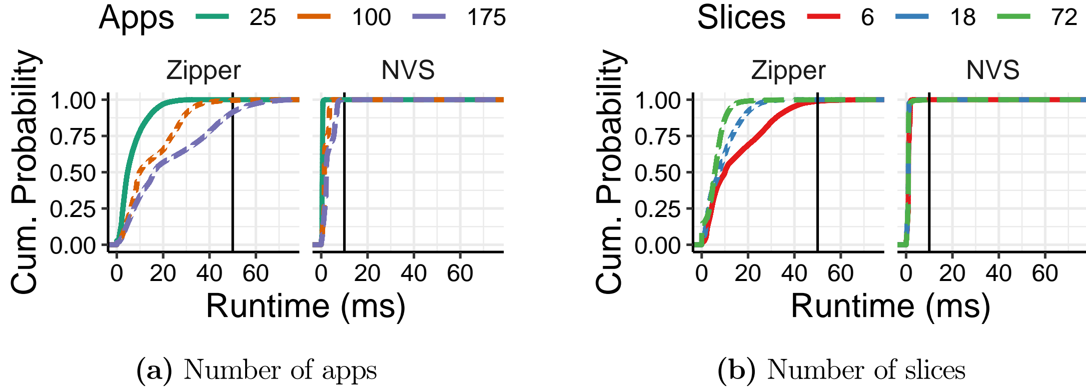
Figure 4-16: Zipper's DNN resource estimator achieves a higher admit rate and utilization by squeezing in apps with lighter demand.

Admit rate

While the penalties are bounded, does the RAN have some unutilized capacity that it could have allocated by admitting more apps? Fig. 4-16a compares the admit rates for different resource availability modules. As we would expect, when the base station is not congested (e.g., 100 apps), the admit rate is high (around 95%) for both the conditional and DNN-based policies. However, as more apps join the system, the DNN policy has a higher admit rate—about 15% higher for 225 apps. We observe similar trends for latency.

The policy that uses the "conditional admit" forecaster rejects apps too aggressively because it conditions its decision on an app's measured penalty in the best effort slice. At higher loads, Zipper ends up allocating most—if not all—spectrum to slices serving apps with SLAs. So Zipper allocates little bandwidth to the best effort slice, and the incoming app receives infrequent air time in its initial 10 second sampling period. Its penalties are thus high, and the controller has little confidence that the app can meet its requirements in the target slice. Moreover, since we keep all rejected apps in the best effort slice, there is further contention for air time, complicating our ability to project the incoming app's performance.

The DNN's admit rate is higher at greater load because it is able to differentiate between different traffic types. For instance, the DNN infers that a voice app could still achieve its light target throughput and latency because the slice's MAC scheduler could accommodate



(a) Number of apps

(b) Number of slices

Figure 4-17: Runtime of Zipper and NVS. Even though Zipper involves more computation than NVS, it is still practical for large workloads.

a new app without degrading the SLAs of the apps already admitted to that slice. By contrast, the "conditional admit" mechanism has little data make this inference, since the voice app gets little air time in a best effort slice.

RAN utilization

Aggressive resource availability forecasts can underutilize the RAN. Fig. 4-16b compares the total RAN throughput for both estimators. Notice that the two methods diverge around 150 apps, after which the DNN can better pack more "lightweight" apps.

4.4.5 Microbenchmarks

We profile Zipper's slice allocation and management overhead as we stress the system with more apps and with more slices. For the traffic distribution listed in Table 4-9, we profile the time it takes to compute the bandwidth allocations and MAC schedules. We compare Zipper with NVS. Fig. 4-17 shows the results as a CDF of runtimes over all scheduling intervals in each 3 minute experiment. The vertical black lines indicate the deadlines required to operate in real time. Zipper, though more complex, reliably meets processing deadlines, as the load increases with more apps or fewer slices (i.e., more apps per slice).

4.5 Discussion

Network APIs

Provisioning connectivity based on app SLAs creates new opportunities. For instance, a developer can split their app into multiple data streams (e.g., audio, video, and sensory for VR), and define SLAs independently for each one. Because Zipper internally estimates network capacity to forecast resource availability, an operator using Zipper could create a network API that exposes metrics like true network capacity to developers. This helps make the network more transparent.

ML components

The ideal deployment for Zipper should have a V100 GPU. However, note that Zipper is compatible with any channel predictor, and Fig. 4-6 shows that a simple EWMA predictor works reasonably well. Moreover, the DNNs in the resource availability estimator are lightweight and do not have real-time deadlines, unlike channel prediction; if resource constrained, operators could serve the DNN on a CPU.

Application adaptivity

An important benefit of the paradigm proposed by Zipper is that application developers no longer need to stress about making their apps reactive to the network. By design, Zipper seeks to provision the right amount of bandwidth so that each app experiences a relatively static network. As a result, interactive and adaptive apps will no longer have to adapt to changing network conditions.

4.6 Conclusion

We developed Zipper, the first 5G RAN slicing system for application-level service assurance. Zipper formulates the scheduling problem with MPC and develops an efficient optimization algorithm to compute SLA-compliant schedules in real-time. Zipper also introduces a primitive to forecast RAN resource availability, with which operators can interface an admission control policy. We implemented Zipper on a production-grade 5G vRAN testbed, adding critical hooks to control slice bandwidths in real time. We evaluated Zipper extensively on realistic workloads, our results showed that Zipper more reliably fulfills app-level SLAs than do QoS schedulers and slice-level service assurance systems.

There several opportunities to extend Zipper. First, a promising future direction is to co-optimize slicing across base stations to provide predictable performance to highly mobile users. Second, we believe we can extend the formulation and optimization techniques we developed to other SLA types, such as energy consumption and bit error rate. Finally, we hope to explore robust economic models for admission control that build on Zipper's resource availability estimator.

Chapter 5

Conclusion

5.1 Summary

This thesis introduces a hierarchical design paradigm to build app-aware resource allocation policies. Our key insight is that a two-level scheduler, where a higher-layer algorithm guides the lower-layer platform scheduler without modifying the latter is the right structure to balance app requirements and the efficiency targets for the system operators. We apply this idea to build systems and algorithms in two domains:

- Mobius allocates tasks from different customers to vehicles in mobility platforms, which are used for food and package delivery, ridesharing, and mobile sensing. Over rounds, Mobius adjusts the inputs to a vehicle routing solver that maximizes task completion throughput to guide the solver to compute schedules that are fair to different customers using the platform. On a trace of Lyft rides in New York City, Mobius computes max-min fair online schedules involving 200 vehicles and over 16,000 tasks, while achieving only 10% less throughput than a classical vehicle routing solver.
- Zipper is a radio resource scheduler that satisfies throughput and latency service-level agreements for apps subscribing to a cellular base station. Zipper bundles apps into network slices, and leverages classical schedulers that maximize base station throughput

to compute SLA-compliant resource schedules for each slice. We show that Zipper reduces tail throughput and latency violations, measured as a ratio of the app's request by 9×, compared to traditional base station schedulers.

5.2 Future Work

Emerging app-level requirements

Both systems in this thesis focused on general metrics like throughput, fairness, and latency. An interesting direction for future work is to extend both systems to consider emerging metrics, such as jitter or power consumption for cellular networks, or rider wait times for mobility platforms. The key challenge is to determine the right mapping from these metrics to inputs into platform schedulers, keeping the rest of the proposed scheduling architecture with throughput-maximizing platform schedulers intact.

Admission control as an architectural component

Zipper (Chapter 4) required an admission control module to reliable fulfill app-level service requirements. However, forecasting resource availability for an incoming app is more generally applicable to other mobile systems. For instance, food delivery operator using Mobius could use a resource availability primitive to determine whether its vehicle fleet has the capacity to accommodate the delivery requirements for a new restaurant that wants to use the service. Future work includes abstracting the resource availability component in Zipper as an architectural component in the two-level scheduler proposed in this thesis.

Monetization preferences

Both systems in this thesis consider performance metrics that affect the reliability and usability of end applications. However, in real-world deployments, applications pay operators for time on the system resources. Future work includes consider app-level objectives in

light of monetization preferences. For instance, an mobile sensing platform may price sensor measurements, or a mobile network operator can charge apps based on bandwidth used in a slice. How do we continue support app-level objectives, considering that different apps may be willing to pay less or more for differentiated connectivity.

Appendix A

Mobius Appendix

A.1 Searching for α-Fair Allocation

§3.5 explains how Mobius generalizes its formulation to support a class of α-fair objectives. Recall that, in each round, the target allocation is the allocation on the convex boundary with the greatest utility U_α; Mobius finds the support allocations that are closest in utility to the target allocation. In each stage of the search, Mobius uses Lagrange Multipliers to identify the face containing the allocation that maximizes U_α. Specifically, for a face described by the equation $\sum_{k \in K} w_k x_k = c$, Mobius computes the allocation $\left(x_1^*, \ldots, x_{|K|}^*\right)$ with greatest utility, subject to the constraint that it lies on the face. The Lagrangian can be written as:

$$\mathcal{L}(\mathbf{x}, \lambda) = U_\alpha(\mathbf{x}) - \lambda \left(\sum_{k \in K} w_k x_k - c \right)$$

where $\mathbf{x} \in \mathbb{R}^{|K|}$. To find the utility-maximizing allocation $\mathbf{x}^*$, we set $\nabla \mathcal{L} = 0$, and solve for $\mathbf{x}^*$:

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial U_\alpha}{\partial x_1^*} - \lambda w_1 \\ \vdots \\ \frac{\partial U_\alpha}{\partial x_k^*} - \lambda w_k \\ \vdots \\ \frac{\partial U_\alpha}{\partial x_{|K|}^*} - \lambda w_{|K|} \\ \sum_{k \in K} w_k x_k^* - c \end{bmatrix} = \mathbf{0}$$

Substituting $\partial U_\alpha / \partial x_k^* = (1/x_k^*)^\alpha$, we get:

$$x_k^* = (\lambda w_k)^{-1/\alpha} \tag{A.1}$$

To solve for λ, we substitute Equation (A.1) into the last element of $\nabla \mathcal{L}$:

$$\lambda = \left[\frac{c}{\sum_{k \in K} w_k^{(1-1/\alpha)}} \right]^{-\alpha} \tag{A.2}$$

Algorithm 1 Mobius Scheduler

```
1: procedure RunMobius(α)
2:     Initialize history x̄∈ℝ^|K|, with x̄=0
3:     for each round do
4:         i ← InitFace()                                   ▷ Use basis weights e_k.
5:         face ← SearchBoundary(α, i)
6:         x* ← argmax_{x∈face} U_α(x+x̄)
7:         Execute schedule with throughput x*.
8:         x̄ ← x̄+x*
9: function SearchBoundary(α, face)
10:    p_ext ← ExtendFace(face)                     ▷ Compute w for face; call VRP.
11:    if p_ext exists then
12:        for p_face ∈ face do
13:            candidate ← {p_ext} + {p∈ face | p≠p_face}
14:            if OptInFace(α, candidate) then
15:                return SearchBoundary(α, candidate)
16:    else
17:        return face
18: function OptInFace(α, face)
19:    opt ← ComputeOpt(α, face)                                ▷ Equation (A.1)
20:    if opt lies within face then
21:        return true
22:    else
23:        return false
```

Mobius computes x_k^* for the $|K|$ faces that arise from the most recent extension, and identifies the one face that contains x_k^* within the boundaries of its vertices. Mobius then extends this face in the next stage. For example, in stage 2 in Fig. 3-6b, we compute x_k^* for faces $\overline{AC}$ and $\overline{CB}$, and continue the search in stage 3 above $\overline{AC}$ because it contains the x_k^* (i.e., it intersects the $y=x$ line).

A.2 Mobius Algorithm

Algorithm 1 provides pseudocode for Mobius's scheduling algorithm. When a mobility platform is initialized, Mobius starts executing the `RunMobius()` function, first initializing the long-term average throughput $\overline{\mathbf{x}}$. In each round, it computes an initial face using the $|K|$ basis weights $\mathbf{e_k}\,\forall k \in K$, where $\mathbf{e_k}$ is a vector of zeros, with the k-th element set to 1.

This gives an initial allocation on every axis in the $|K|$-dimensional customer throughput. Then Mobius runs the `SearchBoundary()` function to compute the face containing the $|K|$ support allocations around the target throughput. It chooses the allocation $\mathbf{x}^*$ that maximizes the total average throughput.

A.3 Optimality of Mobius

§3.4.3 provides intuition about the optimality of Mobius. We show the following results:

1. Mobius gives the optimal solution on the convex boundary in a round.
2. Assuming customer tasks stream in according to a static task arrival model (§3.4.3), Mobius (i) is the optimal allocation on the convex boundary at the end of every around, and (ii) converges to the target allocation with an error that decreases as $\mathcal{O}(1/T)$.

We provide formal mathematical proofs for both results below.

A.3.1 Mobius is Optimal in a Round

We denote H as the set of all possible allocations in a round. We show that for every round, `SearchBoundary()` (Alg. 1, line 5) returns a face on the convex boundary of H that maximizes the utility function U_α. The structure of the proof is as follows:

1. Lemma 1 and Corollary 1 identify that the maximizer of U_α on H is unique.
2. Lemma 2 shows that any point in the extensible region of a face will not lie above other faces.
3. In Lemma 3, we note that extending faces that do not contain the utility maximizing allocation for the stage results in a lower utility.
4. Finally, we piece together these ideas in Theorem 1 in order to prove the optimality of `SearchBoundary()` over one round.

Lemma 1. *For any convex polyhedron $H \in \mathbb{R}^{n^+}$, there is a unique point that maximizes U_α for $\alpha \in (0,\infty]$, and the maximum lies on the boundary of H.*

Proof. U_α is strictly concave for any finite α. When maximizing a concave function over a convex set, the optimum point is unique, the local maximum is the global maximum, and the optimal point is on the boundary of H [15, Theorem 8.3]. □

Corollary 1. *There exists exactly one candidate face at any stage of the convex boundary (among all possible faces in Alg. 1, line 14) for which* `OptInFace()` *is true.*

Proof. It follows from Lemma 1 that the optimal U_α over the current convex boundary is unique. This means that exactly one face must have the optimal within its face. □

Definition 1. *A point* $\mathbf{p}$ *is said to be above (or below) a face described as* $\sum_{k \in K} w_k x_k = c$ *if* $\sum_{k \in K} w_k p_k > c$ *(or* $\sum_{k \in K} w_k p_k < c$*).*

Lemma 2. *Let* $f \in F$ *be a face among all candidate faces* F *during a given stage. Any allocation resulting from an initial call to* `SearchBoundary(f)` *will never lie above any face in* $F \setminus \{f\}$.

Proof. We prove this by contradiction. Suppose $\mathbf{p}$ was above two faces f and $\tilde{f}$. Then there will exist at least one support allocation $\mathbf{x}$ of face $\tilde{f}$ which could not have been found from a previous call to `ExtendFace()`, and $\mathbf{p}$ would have been found instead of $\mathbf{x}$. For example, in Fig. 3-6a, any point above the extensible regions of $\overline{AB}$ and $\overline{BC}$ would contradict the existence of B.

Thus, $\mathbf{p}$ cannot lie above more than one face. This argument is true for any subsequent calls to `SearchBoundary()`, and any subsequent allocation obtained from extending a face has to be below all other faces. □

Lemma 3. *Let* $\mathbf{p}$*, contained in face* f_p*, be the maximizer of* U_α*. The utility of any allocation in the extensible region of a face* $f \neq f_p$ *will be lower than the utility at allocation* $\mathbf{p}$*.*

Proof. Since U_α is concave, and is maximized at point $\mathbf{p}$ on f_p, the value of U_α will only increase if evaluated at a point above f_p. From Lemma 2, we know that every extension of a face other than f_p will be below f_p, and thus have a lower utility than the current value evaluated at $\mathbf{p}$. □

Theorem 1. *In each round,* `SearchBoundary()` *returns the face on the convex boundary that contains the allocation that maximize* U_α*.*

Proof. In every round, `SearchBoundary()` iteratively extends the face that contains the optimal U_α over all faces. Lemma 3 guarantees that any subsequent exploration of a face required by `SearchBoundary()` will only increase U_α, while pruning out the search spaces which cannot improve the solution. When maximizing a concave objective function over a convex set, the local optimal is the global optimal. Therefore, the solution returned when `SearchBoundary()` terminates is the maximum of U_α over H. □

A.3.2 Mobius Converges to the Target Throughput

In our problem setting, customer tasks are only presented to Mobius for the current round, and no knowledge of future task locations is assumed.[1] In this section, we show an interesting result: under the static task arrival model (§3.4.3), Mobius, although myopic, results in allocations that are globally optimal. In other words, asymptotically, Mobius achieves the *same average throughput allocation* that a utility-maximizing oracle, which jointly planned over multiple rounds, would achieve.

Definition 2. *A task distribution is said to be static if the set of throughput allocations (denoted as* H*) is the same for all rounds.*

Definition 3. *The maximum of* U_α *over the convex boundary of* H *is defined as the optimal long term throughput, and is denoted as* $\mathbf{x}^*$*.*

The set of all feasible average throughput allocations after t rounds is denoted as F_t. We prove the following: (i) in every round, Mobius chooses the solution which maximizes $U_\alpha(\overline{\mathbf{x}}(t))$ on the convex boundary of F_t, and (ii) $\overline{\mathbf{x}}(t)$ converges to $\mathbf{x}^*$ with an error that decreases as $1/t$. For brevity, we consider the case with two customers ($|K|=2$). However, the intuition and results generalize for any number of customers. The proof outline is as follows

[1] A greedy approach (discussed in App. A.4) would be a regret-free online algorithm for this planning problem.

1. Lemma 4 and 5 characterize the evolution of the convex boundary
2. Lemma 6 proves that $\mathbf{x}^*$ lies on this boundary.
3. Lemma 7 shows that Mobius is optimal at the end of any finite round t.
4. Finally, we prove asymptotic optimality and describe the rate of convergence in Theorem 2.

Lemma 4. *For any round t, the convex boundary of F_t remains constant.*

Proof. The allocations in F_t are obtained by averaging the throughput obtained over t rounds of H (see Fig. 3-4c). Since H is convex, any average of allocations in H will remain the same boundary. Thus, the convex boundary of H and F_t are the same. □

Lemma 5. *At round t, each face of the convex boundary contains $t-1$ equidistant allocations.*

Proof. Consider one face f in the convex boundary of H. From Lemma 4, we know that the convex boundary of the average throughput at any subsequent round will remain the same (see Fig. 3-4c). However, with subsequent rounds, linear combinations of two corner points that constitute a face will create new allocations that lie along the same face. For example, in Fig. 3-7b, the face $\overline{BE}$ gets "denser" with more allocations, as the number of rounds increases. Remember that in every round, Mobius can only choose between throughput allocations B or E to modify the average. In particular, by round t, n allocations of B and m allocations of E result in an allocation $B_nE_m = \frac{n}{t}B + \frac{m}{t}E$, where n and m are integers. Now, since there are $t-1$ possible combinations of n and m that sum up to t, we get $t-1$ equidistant allocations on the face. □

Lemma 6. $\mathbf{x}^*$ *lies on the face in the convex boundary of H.*

Proof. The maximizer of U_α in the long term searches over the convex boundary of H. Since U_α is a concave function, and the search space is a convex set, $\mathbf{x}^*$ must lie on the face of the convex boundary (Lemma 1). □

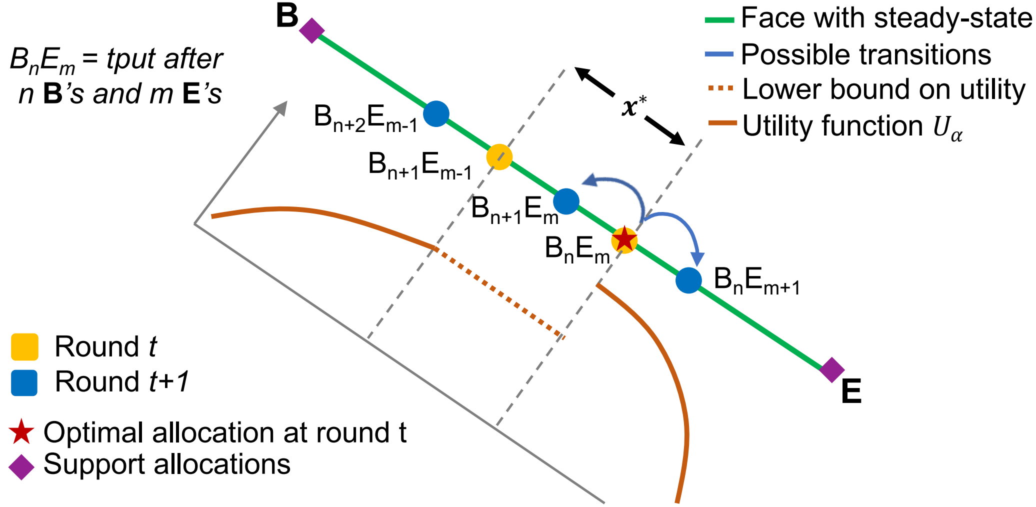


Figure A-1: Proof setup for Lemma 7. The face $\overline{BE}$ is the same as in Fig. 3-7

Lemma 7. *For any round t, Mobius finds the utility-maximal allocation on the convex boundary of F_t.*

Proof. We prove this by induction on the number of rounds. Theorem 1 proves the base case, where $t = 1$. Suppose Mobius finds the highest utility solution on the convex boundary of F_t. We want to show that the schedule that Mobius computes (Algorithm 1) has the highest utility on the convex boundary of F_{t+1}. We use the example in Fig. 3-7b to build this argument. Suppose B_nE_m has the highest utility after $t = n+m$ rounds. Without loss of generality, we assume that the optimal allocation $\mathbf{x}^*$ lies between B_nE_m and $B_{n+1}E_{m-1}$ (rather than B_nE_m and $B_{n-1}E_{m+1}$).

We now argue that Mobius chooses B or E appropriately at round $t+1$ to ensure that the average throughput is the optimal at the end of round $t+1$. Figure A-1 illustrates the setup for this proof.

Recall that U_α is a concave function. Thus, the utility function evaluated over the face $\overline{BE}$ is also concave. This implies that (i) the utility at B_nE_m is higher than any allocation in $(B_nE_m, E]$, and (ii) the utility at $B_{n+1}E_{m-1}$ is higher than any allocation in $[B, B_{n+1}E_{m-1})$. Also, since B_nE_m was the optimal solution at round t (by the inductive hypothesis), the utility at B_nE_m is higher than the utility at $B_{n+1}E_{m-1}$. Thus the utility for any allocation

in $[B_{n+1}E_{m-1}, B_nE_m]$ is lower bounded by the utility at $B_{n+1}E_{m-1}$.

The above relations prove that in round $t+1$, $B_{n+2}E_{m-1}$ can never have the highest utility. Thus the optimal throughput on the convex boundary of F_{t+1} is either $B_{n+1}E_m$ or B_nE_{m+1}. This results in two cases:

- $B_{n+1}E_m$ is the optimal, in which case Mobius would choose allocation B for round $t+1$ to reach the optimal.
- B_nE_{m+1} is the optimal, in which case Mobius would choose allocation E for round $t+1$ to reach the optimal.

Note that by eliminating $B_{n+2}E_{m-1}$ as an optimal allocation, the two remaining candidates can be reached by appropriately choosing B or E in round $t+1$. This concludes the induction argument and proves the lemma. □

Note that Lemma 7 proves that at the end of round t, Mobius not only reaches the best allocation at the end of round t, but it also achieves the best allocation for each preceding round before t.

Theorem 2. *Mobius (i.e. Alg. 1) converges to* $\mathbf{x}^*$ *such that the distance between* $\overline{\mathbf{x}}$ *and* $\mathbf{x}^*$ *decreases as* $\mathcal{O}(1/t)$*, where* t *is the number of rounds.*

Proof. From Lemma 5, we know that at any round t there are $t-1$ feasible cumulative allocations (excluding the extreme points) on a face, and that these allocations split the face into equally-spaced segments of length $\propto \frac{1}{t}$. Thus, the best allocation in F_t converges to the optimal $\mathbf{x}^*$ with an error that is bounded by $\mathcal{O}(1/t)$. Since Lemma 7 establishes that Mobius chooses the optimal allocation in F_t, the result follows. □

A.4 Greedy Heuristic to Maximize U_α

§3.4.4 describes how Mobius builds a suite of warm start schedules to assist the VRP solver in maximizing a weighted sum of customer throughputs. Since Mobius is guided by a utility

function U_α (§3.5), we implement a heuristic that computes an α-fair schedule by performing a greedy maximization of U_α. Note that this algorithm is not guaranteed to result in a schedule on the convex boundary; we instead use it as an initial schedule to warm start the VRP solver (§3.4.4).

The greedy heuristic uses the same formulation as the VRP (§3.1). It computes routes for each vehicle $v \in V$ subject to the budget constraints. Mobius constructs a schedule iteratively; in each iteration, it executes two steps:

1. It constructs an α-fair path *for each vehicle* that meets the budget constraints by trying to maximize U_α.
2. Then, it invokes a VRP solver with the tasks selected by the paths in (1), to build a high throughput schedule to complete the fair allocation of tasks.

At the end of each iteration, it takes the VRP schedule generated by (2) and tries to squeeze more tasks into the path, preserving fairness. It terminates when no new task can be added according to the greedy optimization in step (1). It then runs the VRP one final time, with a very high weight on the final set of α-fair tasks and lower weight on all other customer tasks, so that it can pack the schedule with more tasks to achieve a schedule with high total throughput.

Intuitively, the iterations over steps (1) and (2) create the α-fair schedule with the highest possible throughput, according to the greedy approximation. Then, with the final packing step, we try to boost the throughput of the schedule by fulfilling any additional tasks, without compromising the α-fair allocation we have already committed to.

Before starting to construct a path iteratively, we internally maintain the total number of tasks h_k currently fulfilled by the path, for each customer. To do a greedy maximization of U_α in each iteration of constructing the path, we sort all customer tasks according to the *return-on-investment* R for completing each task. Recall that T_k is the set of tasks requested by customer k and B is the time budget for a round (§3.1). The new throughput

for customer k by fulfilling a task $l \in T_k$ is $x_k = (h_k + 1)/B$. We compute a task $l \in T_k$ as:

$$R(l) = \frac{U_\alpha(\mathbf{x}) - U_\alpha(\mathbf{h})}{c(m,l)} \tag{A.3}$$

where m is the last task in the path and $c(\cdot)$ is the cost to travel from m to l. Then, to select the next task l to add to the path we simply find the task with the greatest return-on-investment:

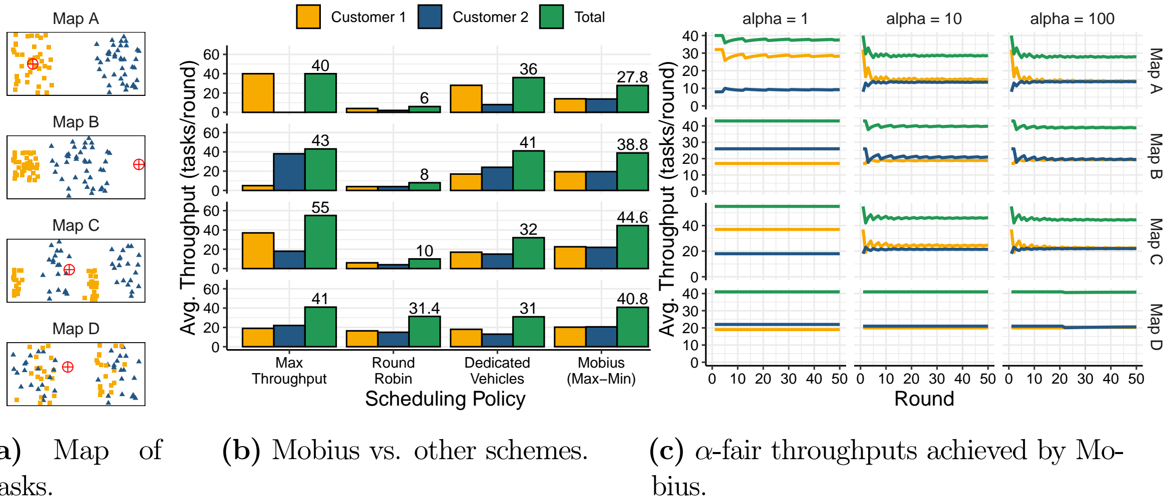
$$\underset{l \in T_k \forall k \in K}{\operatorname{argmax}} \; R(l) \tag{A.4}$$

A.5 Runtime of Mobius

In each round, Mobius uses an efficient algorithm to find support allocations near the target allocation, without having to compute all corner points of the convex hull in each round. This allows Mobius to invoke the VRP solver *sparingly* in order to find an allocation of rates that steers the long-term rates toward the target. We report some highlights from profiling Mobius in Table A-1. These results suggest that the computational overhead for deploying Mobius would be negligible for many mobility applications.

# Cust.	# of Tasks	# of Vehicles	Round Duration (min)	Runtime (s)
2	100	2	10	15
3	150	3	15	20
4	200	4	15	35
6	567	6	90	51
6	999	6	90	59
6	567	24	90	88
6	999	24	90	105

Table A-1: Performance of Mobius on different input sizes.



(a) Map of tasks.

(b) Mobius vs. other schemes.

(c) α-fair throughputs achieved by Mobius.

Figure A-2: Comparing customer throughputs and platform throughput achieved by Mobius and other schemes. Customer tasks stream in according to a static task arrival model. Mobius consistently outperforms other schemes by striking a balance between throughput and fairness.

A.6 Microbenchmarks

We evaluate Mobius on several microbenchmarks involving synthetic customer traces. We vary (1) the spatial distribution of customer tasks, (2) the timescale at which tasks are requested, and (3) the degree α to which a schedule is fair. Customers submit at most 40 tasks, each taking 10 seconds to fulfill, in any round. Between rounds, they renew any fulfilled tasks at the same location. The travel time between any two nodes is based on their Euclidean distance, assuming a constant travel speed of 10 m/s.

A.6.1 Robustness to Spatial Demand

We evaluate Mobius's ability to deliver a fair allocation of customer throughputs, in the presence of highly diverse spatial demand. We construct 4 very different maps (Fig. A-2a), with 2 vehicles starting at $\oplus$. For this experiment, we assume that the customer tasks arrive from a static task arrival model (§3.4.3): the vehicles make a round-trip in each round, and customers renew any fulfilled tasks at the start of every round. Fig. A-2b shows the average

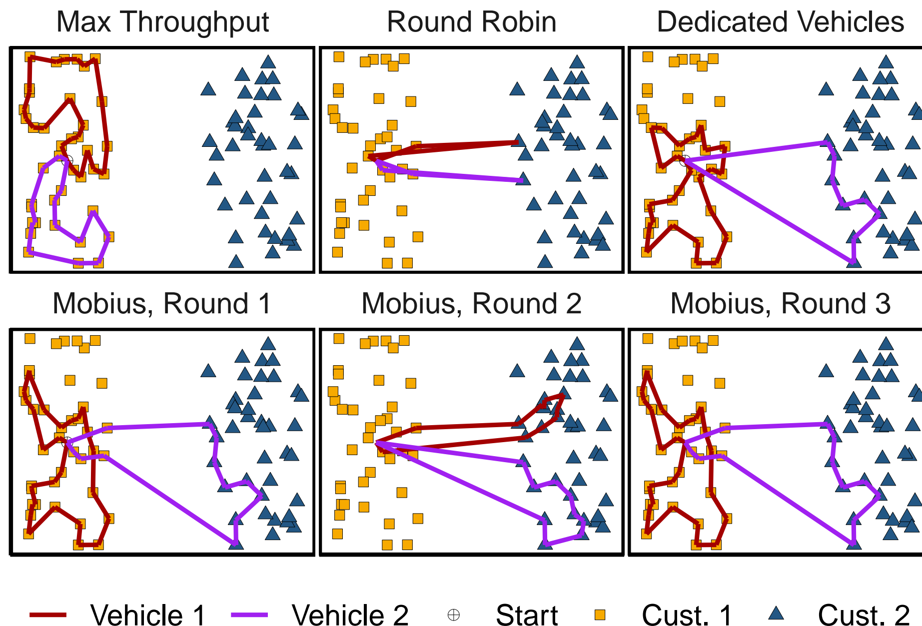


Figure A-3: Snapshot of per-round schedules computed by Mobius (for 3 rounds) and other policies. Mobius compensates for short-term unfairness by switching between schedules on the convex hull over rounds. Other schemes suffer from persistent bias or low throughput.

per-customer and total throughputs achieved by different schemes after 50 rounds.

For all maps, Mobius (with max-min fairness) indeed provides a fair allocation of average customer throughputs. The other baseline schedules exhibit variable performance depending on the task distribution. For example, in Map A, both max throughput and dedicating vehicles achieve dismal fairness. The max throughput schedule only serves customer 1's cluster, and dedicating a vehicle to customer 2 cannot deliver a fair share of throughput, given the round-trip budget constraints. When customers' tasks overlap and have similar spatial density (Map D), the max throughput schedule provides a roughly fair allocation of rates, and the round-robin schedule achieves reasonably high throughput. Dedicating vehicles suffers from poor throughput when there is an incentive to pool tasks from multiple customers into a single vehicle (Map B and Map C).

Focusing on Map A. To illustrate how Mobius converges to fair per-customer allocations without significantly degrading platform throughput, we show in Fig. A-3 schedules

computed for Map A (Fig. A-2a). On the top row, we show max throughput, round-robin, and dedicated schedules; on the bottom row, we show schedules computed by Mobius over 3 consecutive rounds. In round 1, Mobius *exploits a sharing incentive* to pool some of customer 1's tasks into the journey to customer 2's tasks, achieving a similar throughput to the max throughput schedule. By contrast, the max throughput schedule starves customer 2, and the round-robin schedule wastes time moving between clusters.

Mobius is able to compensate for short-term unfairness across multiple rounds of scheduling. Fig. A-3 shows the first 3 schedules that Mobius computed for max-min fairness, assuming fulfilled tasks reappear, as before. Although Mobius does not starve customer 2 in round 1, it still delivers 4× higher throughput to customer 1. However, as we see in round 2, Mobius compensates for this unfairness by prioritizing customer 2, while still exploiting sharing incentive and collecting a few tasks for customer 1 during the round trip. The schedule in round 3 is identical to that in round 1; since tasks arrive according to a static model in this example, the convex hull remains the same across rounds, and so Mobius oscillates between the same two support allocations (schedules). The max throughput and dedicated schedules suffer from a persistent bias in throughput (Fig. A-2b) due to the skew in spatial demand.

A.6.2 Expressive Schedules with α

Mobius's α parameter allows the platform operator to control fairness; with higher α, the platform trades off some total throughput for a fairer allocation of per-customer rates. Fig. A-2c shows, for three different values of α, the long-term throughput for each customer and the platform throughput over time. For all maps (Fig. A-2a), as we increase the degree of fairness α, Mobius compromises some platform throughput. Mobius's scheduler is *expressive*; for instance, if an operator would like high throughput, with the only constraint that no customer gets starved, she can run Mobius with $\alpha = 1$, which ensures a proportionally fair allocation of throughputs. Map A shows an example of this. Furthermore, Mobius

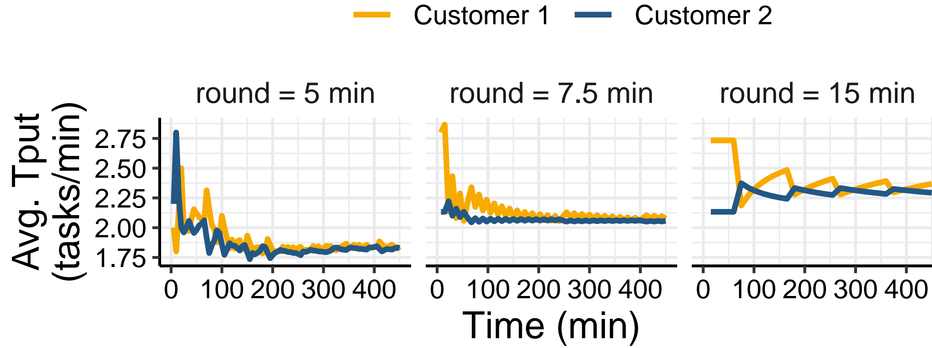


Figure A-4: Mobius converges to the fair allocation of throughputs regardless of the timescale of fairness. Scheduling in shorter rounds converges faster to the fair allocation of rates, but longer round durations lead to schedules with greater platform throughput.

indeed *converges to the target throughput* (§3.4.3); this is best illustrated with the max-min schedules, where the customer throughputs converge to the same rate. Mobius can also converge to any allocation of rates in the spectrum between maximum throughput and max-min fairness (e.g., $\alpha = 10$).

A.6.3 Timescale of Fairness

The duration of a round in Mobius is a parameter; for instance, an operator could set the round length to be the vehicles' fuel time, or the desired timescale of fairness. We expect that, with shorter durations, Mobius can converge faster to a fair allocation. To study this behavior, we consider, in Fig. A-4, max-min fair schedules generated by Mobius on Map A. We consider three round durations (5 mins, 7.5 mins, and 15 mins), requiring the vehicles to return home every 15 minutes, as before. There are three interesting takeaways. First, for all round durations, Mobius provides an equal allocation of rates to both customers. Second, we see that Mobius achieves lower platform throughput for shorter round durations; this is because schedules computed at shorter timescales are more myopic. Third, shorter round durations allow Mobius to converge to faster to an equal allocation rates. In particular, we see that Mobius at a 5-min timescale achieves the fair allocation within 150 minutes, but at a 15-min timescale, it takes nearly 400 minutes to converge. Thus, the timescale of fairness

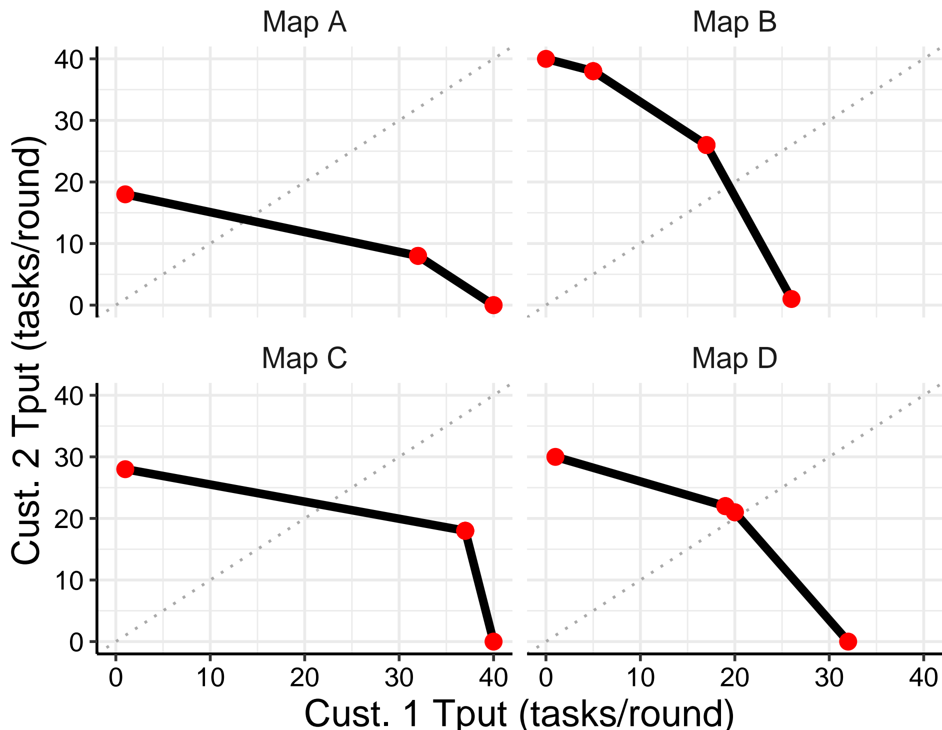


Figure A-5: Convex boundaries computed by Mobius for the different maps shown in Fig. A-2a. The shape of the convex boundary describes the inherent tradeoff between fairness and high throughput.

of fairness allows an operator to trade some total platform throughput for faster convergence.

Additionally, the schedules generated with 5-min and 7.5 min timescales do not observe the static task arrival assumption (§3.4.3). Since the vehicles begin some rounds away from their start locations, the convex hull changes across rounds. Still, Mobius is robust in this setting and provides very similar rates to both customers. The case studies (§3.6.2-§3.6.3) provide more realistic examples where the static task arrival assumption is relaxed.

A.6.4 Geometry of the Convex Boundary

Mobius finds an approximately fair schedule in each round by constructing corner points of the convex boundary. The convex boundary of achievable throughputs succinctly captures the tradeoffs between servicing different customers, based on their spatial demand. Fig. A-5 shows the convex boundaries for four different maps of tasks (shown in Fig. A-2a). We

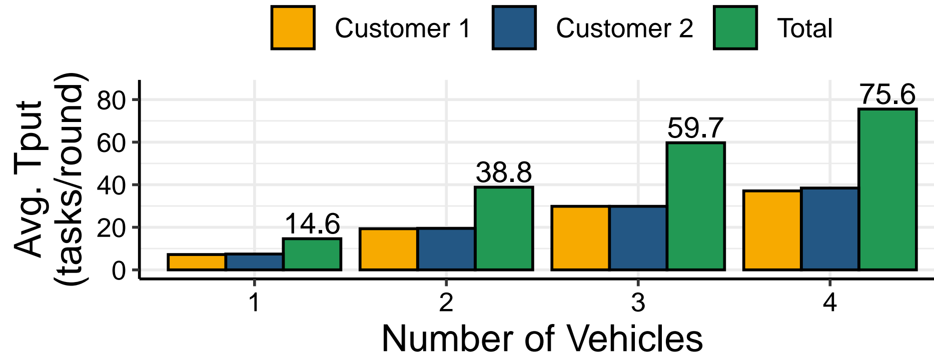


Figure A-6: Long-term per-customer rates computed by Mobius on Map D in Fig. A-2a, for different provisioning of vehicles.

construct the convex boundary using an extended version of Mobius's search boundary. Specifically instead of searching the face containing the utility optimum on the *current* convex boundary, we search *all* faces, i.e., extend the convex boundary in all directions. The terminating conditions remain the same: we know we have reached face on the convex boundary when we cannot extend it further.

The geometry of the convex boundary indicates which customers the platform can service more easily. For instance, notice that the boundaries for Map A and Map C are both skewed toward customer 1, since the platform incurs less overhead to service both customers. In contrast, the convex boundary for Map D is symmetric, since each customer is equally easy to service.

A.6.5 Varying the Number of Vehicles

The results in §3.6 show that dedicating vehicles can (i) miss out on sharing incentive, leading to lower platform throughput, and (ii) lead to unfairness in situations where it is inherently harder to service some customers. However, dedicating vehicles is only a viable policy when the number of vehicles is a multiple of the number of customers. Mobius makes no assumptions about the number of vehicles in the mobility platform; in this section, we study the performance of Mobius in a mobility platform with different numbers of vehicles.

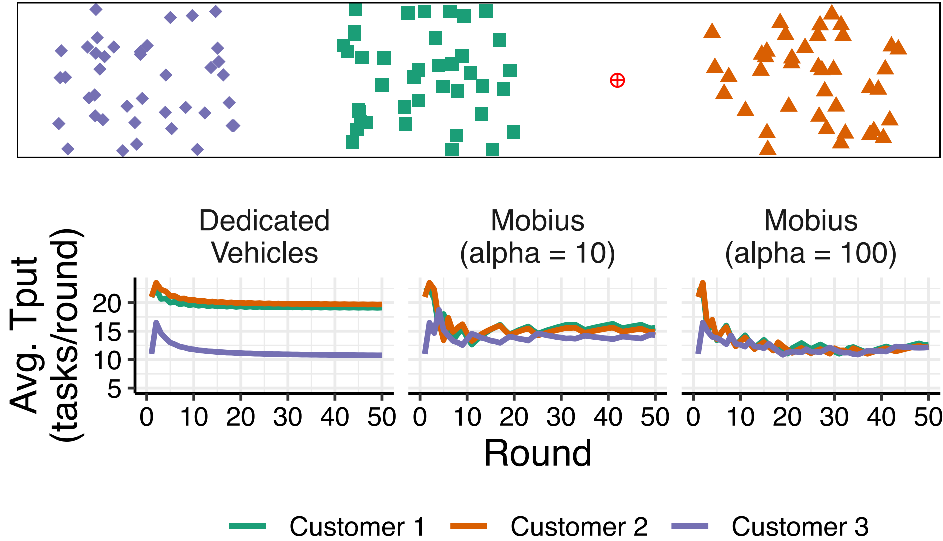


Figure A-7: Mobius vs. dedicating vehicles for example with 3 customers. Mobius converges to a fair allocation of rates for customers, when the assumption on static task arrival is relaxed.

Fig. A-6 shows the per-customer average throughputs $\overline{x}_k(t)$ achieved by Mobius (for max-min fairness) on Map B (Fig. A-2a) for different numbers of vehicles. In all cases, Mobius converges to a max-min fair allocation of rates. As expected, the throughput of the platform increases with more vehicles, since the platform can complete more in parallel.

A.6.6 A Case with Three Customers

§A.6 showed a controlled study of the properties of Mobius, in environments with two customers. Fig. A-7 shows an example with three customers and three vehicles, all starting at $\oplus$. We let customers renew fulfilled tasks after every round. We consider a fairness timescale of 5 minutes, and require that the vehicles return home every 15 minutes (i.e., 3 rounds); so we relax the assumption on static task arrival, i.e., the convex boundary is identical every 3 rounds. Fig. A-7 shows time series chart of per-customer long-term throughputs achieved by Mobius (for $\alpha = 10$ and $\alpha = 100$) and by dedicating vehicles. The schedule that dedicates vehicles to customers misses out on the opportunity to fulfill tasks for customer 1 on the way to customer 3's cluster. Additionally, notice that Mobius can

provide a fair allocation of rates for 3 customers, and α allows Mobius to control the degree to which the rates converge to the same value.

Appendix B

Zipper Appendix

B.1 Allocating Slice Bandwidth in Zipper

B.1.1 Forecasting the Wireless Channel with an RNN

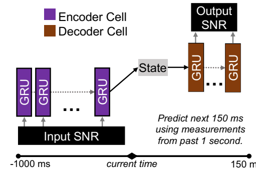


Figure B-1: Architecture of RNN model to forecast wireless channel.

To forecast each user's channel, we train a sequence-to-sequence Recurrent Neural Network (RNN) [105], which uses an input sequence of SNR measurements over the last 1 second to predict a sequence of SNR measurements over the next 150 milliseconds. Each RNN cell is a Gated Recurrent Unit (GRU) [31], a lightweight mechanism that learns both short-term and long-term trends in a signal. GRUs are popular in temporal prediction tasks, like time-series prediction [63] and natural language models [126]. Zipper's RNN model includes two types of GRU cells—encoder and decoder—allowing the model to develop two distinct skills: (i) to build a model of the current state by looking at past values and (ii) to understand the current state to predict future values. Our implementation uses 50 hidden neurons in each layer of the encoder and decoder. Fig. B-1 illustrates the architecture of this RNN.

RNNs are emerging a popular method to forecast timeseries, including wireless channel [69, 79, 83, 91]. We develop and train a model, but note that Zipper can support any predictor of wireless channel. §4.2.1 characterizes the requirements for suitable predictor.

B.1.2 Monotonicity of Throughput and Latency

An app's instantaneous RAN throughput depends on (i) the number of resource blocks it is allocated in each slot and (ii) the MCS scheme used to modulate data onto those resource blocks [9, 77]. If a scheduler assigns an app more resource blocks (i.e., bandwidth) in a slot, then the app will experience a higher throughput. Therefore, app throughput is a monotonically-increasing function of slice bandwidth.

§4.2.2 explains how latency is a monotonically-decreasing function of slice bandwidth. The intuition is that adding more bandwidth to a slice gives the scheduler more space to fit packets for an app and therefore reduce its latency.

B.1.3 Algorithm

Algorithm 2 specifies (in pseudocode) how Zipper computes slice bandwidth allocations. `SearchBandwidth()` is a recursive function that evaluates candidate bandwidths, pruning the search space with binary search, using the property that app throughput and latency vary monotonically with slice bandwidth.

B.2 Estimating Resource Availability in Zipper

B.2.1 DNN Architecture

Zipper builds a family of DNNs to estimate resource availability. Each DNN caters to different slice types. The input embedding consists of (i) the number of apps in the slice (including the incoming app, if applicable) and (ii) the number of apps in each SNR bucket.

Algorithm 2 Allocating slice bandwidth in Zipper

```
1: procedure ZIPPER
2:     for each scheduling round t do
3:         for slice s∈S do
4:             B_s ← FINDBANDWIDTH(s, B)
5:         Resolve conflict if ∑_{s∈S} B_s > B
6:         Update app x̄_a(t) and d̄_a(t) and run MAC/PHY
7: function FINDBANDWIDTHSLICE(s, B)
8:     Grab snapshot of app queues, throughput, and latency from s
9:     Forecast SNR for apps in s
10:    return SEARCHBANDWIDTH(s, 0, B, NULL)
11: function SEARCHBANDWIDTH(s, B_min, B_max, best)
12:    B̃_s ← (B_min + B_max)/2                          ▷ Find midpoint bandwidth.
13:    schedule ← RUNMAC(s, B̃_s)
14:    throughput ← ISTHROUGHPUTVALID(schedule)
15:    latency ← ISLATENCYVALID(schedule)
16:    if throughput ∧ latency then
17:        best = B̃_s                                    ▷ Save best bandwidth.
18:        decrease = true
19:    if B̃_s ≤ B_min or B̃_s ≥ B_max then
20:        return best
21:    if decrease then
22:        return SEARCHBANDWIDTH(s, B_min, B̃_s, best)
23:    else
24:        return SEARCHBANDWIDTH(s, B̃_s, B_max, best)
```

There are four possible SNR buckets.

Each DNN is a fully-connected network with 5 hidden layers, ranging from 512 to 10 neurons. The final layer has 7 output values for different percentiles of the probability distribution, i.e., p10, p25, p50, p75, p90, p95, p99.

Bibliography

[1] Adafruit. Pm2.5 air quality sensor. https://learn.adafruit.com/pm25-air-quality-sensor.

[2] Anwer Al-Dulaimi, Xianbin Wang, and Chih-Lin I. *Network Slicing for 5G Networks*, pages 327–370. John Wiley & Sons, New Jersey, USA, 2018.

[3] Robert S. Allison, Joshua M. Johnston, Gregory Craig, and Sion Jennings. Airborne optical and thermal remote sensing for wildfire detection and monitoring. *Sensors*, 16(8):1310, 2016.

[4] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proc. Natl. Acad. Sci. USA*, 114(3):462–467, 2017.

[5] Altran. Capgemini altran. Technical report, Capgemini, 2023.

[6] Akarshani Amarasinghe, Chathura Suduwella, Charith Elvitigala, Lasith Niroshan, Rangana Jayashanka Amaraweera, Kasun Gunawardana, Prabash Kumarasinghe, Kasun De Zoysa, and Chamath Keppetiyagama. A machine learning approach for identifying mosquito breeding sites via drone images. In M. Rasit Eskicioglu, editor, *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys 2017, Delft, Netherlands, November 06-08, 2017*, pages 68:1–68:2. ACM, 2017.

[7] Sihem Bakri, Pantelis A. Frangoudis, Adlen Ksentini, and Maha Bouaziz. Data-driven RAN slicing mechanisms for 5G and beyond. *IEEE Transactions on Network and Service Management*, 18(4):4654–4668, 2021.

[8] Egon Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[9] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. Detecting if LTE is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Arjun Balasingam, Karthik Gopalakrishnan, Radhika Mittal, Venkat Arun, Ahmed Saeed, Mohammad Alizadeh, Hamsa Balakrishnan, and Hari Balakrishnan.

Throughput-fairness tradeoffs in mobility platforms. https://arxiv.org/abs/2105.11999, 2021.

[11] Arjun Balasingam, Karthik Gopalakrishnan, Radhika Mittal, Venkat Arun, Ahmed Saeed, Mohammad Alizadeh, Hamsa Balakrishnan, and Hari Balakrishnan. Throughput-fairness tradeoffs in mobility platforms. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 363–375, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Arjun Balasingam, Manikanta Kotaru, and Paramvir Bahl. Application-level service assurance with 5G RAN slicing. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, USA, April 16-18, 2024*. USENIX Association, 2024.

[13] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 173–188, Oakland, CA, May 2015. USENIX Association.

[14] Nimantha Baranasuriya, Vishnu Navda, Venkata N. Padmanabhan, and Seth Gilbert. QProbe: Locating the bottleneck in cellular communication. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Amir Beck. *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with MATLAB*, chapter Chapter 8: Convex Optimization, pages 147–168. SIAM, 2014.

[16] Dimitris Bertsimas, Patrick Jaillet, and Sébastien Martin. Online vehicle routing: The edge of optimization in large-scale applications. *Oper. Res.*, 67(1):143–162, 2019.

[17] Dimitris J. Bertsimas. A vehicle routing problem with stochastic demand. *Oper. Res.*, 40(3):574–585, 1992.

[18] Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. Scope: An open and softwarized prototyping platform for nextg systems. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 415–426, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*, chapter Convex Sets, page 21–66. Cambridge University Press, 2004.

[20] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*, chapter Convex Optimization Problems, pages 146–148. Cambridge University Press, 2004.

[21] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[22] Anton Braverman, J. G. Dai, Xin Liu, and Lei Ying. Empty-car routing in ridesharing systems. *Oper. Res.*, 67(5):1437–1452, 2019.

[23] Nishant Budhdev, Raj Joshi, Pravein Govindan Kannan, Mun Choon Chan, and Tulika Mitra. Fsa: Fronthaul slicing architecture for 5G using dataplane programmable switches. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 723–735, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Pablo Caballero, Albert Banchs, Gustavo de Veciana, and Xavier Costa-Pérez. Multi-tenant radio access network slicing: Statistical multiplexing of spatial loads. *IEEE/ACM Transactions on Networking*, 25(5):3044–3058, 2017.

[25] Pablo Caballero, Albert Banchs, Gustavo de Veciana, Xavier Costa-Pérez, and Arturo Azcorra. Network slicing for guaranteed rate services: Admission control and resource allocation games. *IEEE Transactions on Wireless Communications*, 17(10):6419–6432, 2018.

[26] Zizheng Cao, Qian Ma, Adrianus Bernardus Smolders, Yuqing Jiao, Michael J. Wale, Chin Wan Oh, Hequan Wu, and Antonius Marcellus Jozef Koonen. Advanced integration techniques on broadband millimeter-wave beam steering for 5G wireless networks and beyond. *IEEE Journal of Quantum Electronics*, 52(1):1–20, 2016.

[27] Francesco Capozzi, Giuseppe Piro, Luigi Alfredo Grieco, Gennaro Boggia, and Pietro Camarda. Downlink packet scheduling in LTE cellular networks: Key design issues and a survey. *IEEE communications surveys & tutorials*, 15(2):678–700, 2012.

[28] Mohammed Chahbar, Gladys Diaz, Abdulhalim Dandoush, Christophe Cérin, and Kamal Ghoumid. A comprehensive survey on the E2E 5G network slicing model. *IEEE Transactions on Network and Service Management*, 18(1):49–62, 2020.

[29] Chia-Yu Chang and Navid Nikaein. RAN runtime slicing system for flexible and dynamic service execution environment. *IEEE Access*, 6:34018–34042, 2018.

[30] Yongzhou Chen, Ruihao Yao, Haitham Hassanieh, and Radhika Mittal. Channel-aware 5G RAN slicing with customizable schedulers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[31] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[32] NYC Taxi & Limousine Commission. Taxi & limousine commission - homepage. https://www1.nyc.gov/site/tlc/index.page.

[33] NYC Taxi & Limousine Commission. TLC trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[34] Estefania Coronado and Roberto Riggio. Flow-based network slicing: Mapping the future mobile radio access networks. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2019.

[35] Intel Corporation. FlexRAN reference architecture for wireless. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, 2022.

[36] Xavier Costa-Pérez, Joerg Swetina, Tao Guo, Rajesh Mahindra, and Sampath Rangarajan. Radio access network virtualization for future mobile carrier networks. *IEEE Communications Magazine*, 51(7):27–35, 2013.

[37] X. de Foy. Network slicing – 3GPP use case. Technical report, Internet Engineering Task Force, 2017.

[38] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In Lawrence H. Landweber, editor, *SIGCOMM '89, Proceedings of the ACM Symposium on Communications Architectures & Protocols, Austin, TX, USA, September 19-22, 1989*, pages 1–12. ACM, 1989.

[39] Ashutosh Dhekne, Ayon Chakraborty, Karthikeyan Sundaresan, and Sampath Rangarajan. TrackIO: Tracking first responders inside-out. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 751–764. USENIX Association, 2019.

[40] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. *Agora: Real-Time Massive MIMO Baseband Processing in Software*, page 232–244. Association for Computing Machinery, New York, NY, USA, 2020.

[41] DJI. DJI - official website. https://www.dji.com/.

[42] DJI. Flame wheel ARF kit: Multirotor flying platform for entertaining and amateur ap. https://www.dji.com/flame-wheel-arf.

[43] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. Iperf. https://iperf.fr/, 2022.

[44] Yvan Dumas, Jacques Desrosiers, and François Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54(1):7–22, 1991.

[45] Ericsson. Ericsson mobility report. Technical report, Ericsson, 2021.

[46] Jonathan C. Las Fargeas, Pierre T. Kabamba, and Anouck R. Girard. Cooperative surveillance and pursuit using unmanned aerial vehicles and unattended ground sensors. *Sensors*, 15(1):1365–1388, 2015.

[47] FlytBase. Flytos: Operating system for drones. https://flytbase.com/flytos/.

[48] Xenofon Foukas, Mahesh K. Marina, and Kimon Kontovasilis. Orion: RAN slicing for a flexible and cost-effective multi-service mobile network architecture. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 127–140, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5G RAN analytics and control to a new level. In *Technical Report*, December 2022.

[50] Carlos E Garcia, David M Prett, and Manfred Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[51] Krishna C. Garikipati, Kassem Fawaz, and Kang G. Shin. RT-OPEX: Flexible scheduling for cloud-RAN processing. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 267–280, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Yasaman Ghasempour, Muhammad Kumail Haider, and Edward W. Knightly. Decoupling beam steering and user selection for mu-mimo 60-ghz wlans. *IEEE/ACM Transactions on Networking*, 26(5):2390–2403, 2018.

[53] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[54] B.L. Golden, S. Raghavan, and E.A. Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Operations Research/Computer Science Interfaces Series. Springer US, 2008.

[55] Google, Inc. Google maps platform | distance matrix api. https://developers.google.com/maps/documentation/distance-matrix/overview, 2020.

[56] Tao Guo and Alberto Suárez. Enabling 5G RAN slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[57] Tao Guo and Alberto Suárez. Enabling 5G RAN slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[58] Gurobi Optimization, LLC. Gurobi optimizer reference manual. http://www.gurobi.com", 2020.

[59] Songtao He, Favyen Bastani, Arjun Balasingam, Karthik Gopalakrishnan, Ziwen Jiang, Mohammad Alizadeh, Hari Balakrishnan, Michael J. Cafarella, Tim Kraska, and Sam Madden. Beecluster: drone orchestration via predictive optimization. In Eyal de Lara, Iqbal Mohomed, Jason Nieh, and Elizabeth M. Belding, editors, *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, pages 299–311. ACM, 2020.

[60] Alexander Van't Hof and Jason Nieh. Androne: Virtual drone computing in the cloud. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 6:1–6:16. ACM, 2019.

[61] Robert V Hogg and Allen T Craig. Introduction to mathematical statistics.(5"" edition). *Englewood Hills, New Jersey*, 1995.

[62] Yan Huang, Shaoran Li, Y. Thomas Hou, and Wenjing Lou. Gpf: A gpu-based design to achieve 100 μs scheduling for 5G nr. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 207–222, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Michael Hüsken and Peter Stagge. Recurrent neural networks for time series classification. *Neurocomputing*, 50:223–235, 2003.

[64] Chih-Lin I and Sachin Katti. O-RAN: Towards an open and smart RAN. Technical report, Open RAN Alliance, 2018.

[65] IBM. Ibm cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer, 2021.

[66] European Telecommunications Standards Institute. *Physical layer procedures for data*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[67] European Telecommunications Standards Institute. *System Architecture for the 5G System*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[68] Menglan Jiang, Massimo Condoluci, and Toktam Mahmoodi. Network slicing management & prioritization in 5G mobile systems. In *European Wireless 2016; 22th European Wireless Conference*, pages 1–6, 2016.

[69] Wei Jiang and Hans Dieter Schotten. Deep learning for fading channel prediction. *IEEE Open Journal of the Communications Society*, 1:320–332, 2020.

[70] Nicolas Jozefowiez, Frédéric Semet, and El-Ghazali Talbi. Multi-objective vehicle routing problems. *Eur. J. Oper. Res.*, 189(2):293–309, 2008.

[71] Vijay Karamcheti and Andrew A Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 6–6. IEEE, 1998.

[72] Mohammad T Kawser, Nafiz Imtiaz Bin Hamid, Md Nayeemul Hasan, M Shah Alam, and M Musfiqur Rahman. Downlink snr to cqi mapping for different multipleantenna techniques in lte. *International journal of information and electronics engineering*, 2(5):757, 2012.

[73] Frank Kelly. Fairness and stability of end-to-end congestion control. *Eur. J. Control*, 9(2-3):159–176, 2003.

[74] Frank P. Kelly, A. K. Maulloo, and David Kim Hong Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *J. Oper. Res. Soc.*, 49(3):237–252, 1998.

[75] Behnam Khodapanah, Ahmad Awada, Ingo Viering, David Oehmann, Meryem Simsek, and Gerhard P. Fettweis. Fulfillment of service level agreements via slice-aware radio resource management in 5G networks. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–6, 2018.

[76] Ravi Kokku, Rajesh Mahindra, Honghai Zhang, and Sampath Rangarajan. Nvs: A substrate for virtualizing wireless resources in cellular networks. *IEEE/ACM Transactions on Networking*, 20(5):1333–1346, 2012.

[77] Swarun Kumar, Ezzeldin Hamed, Dina Katabi, and Li Erran Li. LTE radio analytics made easy and accessible. *SIGCOMM Comput. Commun. Rev.*, 44(4):211–222, aug 2014.

[78] HyunJong Lee, Shadi Noghabi, Brian Noble, Matthew Furlong, and Landon Cox. Bumblebee: Application-aware adaptation for edge-cloud orchestration. In *Symposium on Edge Computing*. ACM/IEEE, December 2022.

[79] Jinsung Lee, Sungyong Lee, Jongyun Lee, Sandesh Dhawaskar Sathyanarayana, Hyoyoung Lim, Jihoon Lee, Xiaoqing Zhu, Sangeeta Ramakrishnan, Dirk Grunwald, Kyunghan Lee, and Sangtae Ha. Perceive: Deep learning-based cellular uplink prediction using real-time scheduling patterns. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 377–390, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Junling Li, Weisen Shi, Peng Yang, Qiang Ye, Xuemin Sherman Shen, Xu Li, and Jaya Rao. A hierarchical soft RAN slicing framework for differentiated service provisioning. *IEEE Wireless Communications*, 27(6):90–97, 2020.

[81] Xin Li, Chengcheng Guo, Lav Gupta, and Raj Jain. Efficient and secure 5G core network slice provisioning based on vikor approach. *IEEE Access*, 7:150517–150529, 2019.

[82] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2005.

[83] Changqing Luo, Jinlong Ji, Qianlong Wang, Xuhui Chen, and Pan Li. Channel state information prediction for 5G wireless communications: A deep learning approach. *IEEE Transactions on Network Science and Engineering*, 7(1):227–236, 2018.

[84] Sara Mahmoud, Nader Mohamed, and Jameela Al-Jaroodi. Integrating UAVs into the cloud using the concept of the web of things. *J. Robotics*, 2015:631420:1–631420:10, 2015.

[85] Wenguang Mao, Zaiwei Zhang, Lili Qiu, Jian He, Yuchen Cui, and Sangki Yun. Indoor follow me drone. In Tanzeem Choudhury, Steven Y. Ko, Andrew Campbell, and Deepak Ganesan, editors, *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*, pages 345–358. ACM, 2017.

[86] Vera Mersheeva and Gerhard Friedrich. Multi-uav monitoring with priorities and limited energy resources. In Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*, pages 347–356. AAAI Press, 2015.

[87] Microsoft. Azure private 5G core. Technical report, Microsoft, 2023.

[88] Scott Middleton. Discrimination, Regulation, and Design in Ridehailing. Master's thesis, Massachusetts Institute of Technology, 5 2018.

[89] Jose Carlos Molina, Ignacio Eguia, Jesus Racero, and Fernando Guerrero. Multi-objective vehicle routing problem with cost and emission functions. *Procedia - Social and Behavioral Sciences*, 160:254–263, 2014. XI Congreso de Ingenieria del Transporte (CIT 2014).

[90] Luca Mottola, Mattia Moretta, Kamin Whitehouse, and Carlo Ghezzi. Team-level programming of drone sensor networks. In Ákos Lédeczi, Prabal Dutta, and Chenyang Lu, editors, *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, SenSys '14, Memphis, Tennessee, USA, November 3-6, 2014*, pages 177–190. ACM, 2014.

[91] L Srikar Muppirisetty, Tommy Svensson, and Henk Wymeersch. Spatial wireless channel prediction under location uncertainty. *IEEE Transactions on Wireless Communications*, 15(2):1031–1044, 2015.

[92] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 188–201. ACM, 2016.

[93] Nidal Nasser, Lutful Karim, and Tarik Taleb. Dynamic multilevel priority packet scheduling scheme for wireless sensor network. *IEEE transactions on wireless communications*, 12(4):1448–1459, 2013.

[94] Nvidia. Nvidia aerial SDK. https://developer.nvidia.com/aerial-sdk, 2022.

[95] O-RAN. O-RAN specifications. Technical report, O-RAN Alliance, 2023.

[96] Mourice O. Ojijo and Olabisi E. Falowo. A survey on slice admission control strategies and optimization schemes in 5G network. *IEEE Access*, 8:14977–14990, 2020.

[97] Arled Papa, Alba Jano, Serkut Ayvaşık, Onur Ayan, H. Murat Gürsu, and Wolfgang Kellerer. User-based quality of service aware multi-cell radio access network slicing. *IEEE Transactions on Network and Service Management*, 19(1):756–768, 2022.

[98] Laurent Perron and Vincent Furnon. Or-tools. https://developers.google.com/optimization/routing/vrp.

[99] Riccardo Petrolo, Yingyan Lin, and Edward W. Knightly. ASTRO: autonomous, sensing, and tetherless networked drones. In *Proceedings of the 4th ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications, DroNet@MobiSys 2018, Munich, Germany, June 10-15, 2018*, pages 1–6. ACM, 2018.

[100] Qulsar. Qulsar Qg2. https://qulsar.com/Products/Systems/Qg_2.html, 2022.

[101] Darijo Raca, Dylan Leahy, Cormac J. Sreenan, and Jason J. Quinlan. Beyond throughput, the next generation: A 5G dataset with channel and context metrics. In *Proceedings of the 11th ACM Multimedia Systems Conference*, MMSys '20, page 303–308, New York, NY, USA, 2020. Association for Computing Machinery.

[102] Carl Edward Rasmussen. Gaussian processes in machine learning. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning, ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures*, volume 3176 of *Lecture Notes in Computer Science*, pages 63–71. Springer, 2003.

[103] Darshan A. Ravi, Vijay K. Shah, Chengzhang Li, Y. Thomas Hou, and Jeffrey H. Reed. RAN slicing in multi-MVNO environment under dynamic channel conditions. *IEEE Internet of Things Journal*, 9(6):4748–4757, 2022.

[104] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[105] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[106] Oriol Sallent, Jordi Perez-Romero, Ramon Ferrus, and Ramon Agusti. On radio access network slicing from a radio resource management perspective. *IEEE Wireless Communications*, 24(5):166–174, 2017.

[107] Vincenzo Sciancalepore, Konstantinos Samdanis, Xavier Pérez Costa, Dario Bega, Marco Gramaglia, and Albert Banchs. Mobile traffic forecasting for maximizing 5G network slicing resource utilization. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9, Atlanta, GA, 2017. IEEE.

[108] Nashid Shahriar, Sepehr Taeb, Shihabur Rahman Chowdhury, Mubeen Zulfiqar, Massimo Tornatore, Raouf Boutaba, Jeebak Mitra, and Mahdi Hemmati. Reliable slicing of 5G transport networks with bandwidth squeezing and multi-path provisioning. *IEEE Transactions on Network and Service Management*, 17(3):1418–1431, 2020.

[109] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information processing letters*, 84(2):93–98, 2002.

[110] Cisco Systems. Cisco annual internet report (2018 - 2023). Technical report, Cisco Systems, 2020.

[111] Éric D. Taillard. Parallel iterative search methods for vehicle routing problems. *Networks*, 23(8):661–673, 1993.

[112] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, jan 2011.

[113] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem*, volume 9 of *SIAM monographs on discrete mathematics and applications*. SIAM, 2002.

[114] Sebastian Troia, Andres Felipe Rodriguez Vanegas, Ligia Maria Moreira Zorello, and Guido Maier. Admission control and virtual network embedding in 5G networks: A deep reinforcement-learning approach. *IEEE Access*, 10:15860–15875, 2022.

[115] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, Cambridge, UK, 2005.

[116] Uber Technologies. What is destination discrimination? https://help.uber.com/driving-and-delivering/article/what-is-destination-discrimination?nodeId=9bde02cc-3d43-4837-9384-d28c57755fd9, 2021.

[117] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N. Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An IoT platform for data-driven agriculture. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 515–529. USENIX Association, 2017.

[118] Mohammad M. Vazifeh, Paolo Santi, Giovanni Resta, Steven H. Strogatz, and Carlo Ratti. Addressing the minimum fleet problem in on-demand urban mobility. *Nat.*, 557(7706):534–538, 2018.

[119] Melissa Vetromille, Luciano Ost, César AM Marcon, Carlos Reif, and Fabiano Hessel. Rtos scheduler implementation in hardware and software for real time applications. In *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, pages 163–168. IEEE, 2006.

[120] Pierre-Brice Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *2006 6th IEEE-RAS International Conference on Humanoid Robots, Genova, Italy, December 4-6, 2006*, pages 137–142, Genova, Italy, 2006. IEEE.

[121] Jun Wu, Zhifeng Zhang, Yu Hong, and Yonggang Wen. Cloud radio access network (c-RAN): a primer. *IEEE network*, 29(1):35–41, 2015.

[122] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, feb 2020. USENIX Association.

[123] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. Bigstation: Enabling scalable real-time signal processingin large mu-mimo systems. *SIGCOMM Comput. Commun. Rev.*, 43(4):399–410, aug 2013.

[124] Justin Yapp, Remzi Seker, and Radu F. Babiceanu. UAV as a service: A network simulation environment to identify performance and security issues for commercial UAVs in a coordinated, cooperative environment. In Jan Hodický, editor, *Modelling and Simulation for Autonomous Systems - Third International Workshop, MESAS*

2016, Rome, Italy, June 15-16, 2016, Revised Selected Papers, volume 9991 of *Lecture Notes in Computer Science*, pages 347–355, 2016.

[125] Qiang Ye, Junling Li, Kaige Qu, Weihua Zhuang, Xuemin Sherman Shen, and Xu Li. End-to-end quality of service in 5G networks: Examining the effectiveness of a network slicing framework. *IEEE Vehicular Technology Magazine*, 13(2):65–74, 2018.

[126] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.

[127] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 325–338, New York, NY, USA, 2015. Association for Computing Machinery.

[128] Wanke Yu, Chunhui Zhao, and Biao Huang. Stationary subspace analysis-based hierarchical model for batch processes monitoring. *IEEE Transactions on Control Systems Technology*, 29(1):444–453, 2020.

[129] Yasir Zaki, Thushara Weerawardane, Carmelita Görg, and Andreas Timm-Giel. Multi-qos-aware fair scheduling for LTE. In *Proceedings of the 73rd IEEE Vehicular Technology Conference, VTC Spring 2011, 15-18 May 2011, Budapest, Hungary*, pages 1–5, Budapest, Hungary, 2011. IEEE.

[130] Lanfranco Zanzi, Vincenzo Sciancalepore, Andres Garcia-Saavedra, Hans Dieter Schotten, and Xavier Costa-Pérez. Laco: A latency-driven network slicing orchestration in beyond-5G networks. *IEEE Transactions on Wireless Communications*, 20(1):667–682, 2021.

[131] David Zipper. Did uber just enable discrimination by destination? https://www.bloomberg.com/news/articles/2019-12-11/the-discrimination-risk-in-uber-s-new-driver-rule, 2019.