Application-Level Service Assurance with 5G RAN Slicing

Arjun Balasingam, *MIT CSAIL;* Manikanta Kotaru and Paramvir Bahl, *Microsoft*

https://www.usenix.org/conference/nsdi24/presentation/balasingam

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Application-Level Service Assurance with 5G RAN Slicing

Arjun Balasingam
MIT CSAIL

Manikanta Kotaru
Microsoft

Paramvir Bahl
Microsoft

Abstract

This paper presents Zipper, a novel Radio Access Network (RAN) slicing system that provides assurances of application-level throughput and latency. Existing RAN slicing systems optimize for slice-level assurance, but these methods fail to provide predictable network performance to individual mobile apps. Extending the slice-level formulation to app-level introduces an intractable optimization problem with exploding state and action spaces. To simplify the search space, Zipper casts the problem as a model predictive controller, and explicitly tracks the network dynamics of each user. It uses an efficient algorithm to compute slice bandwidth allocations that meet each app's requirements. 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 we 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 Introduction

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* [20], 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[1] with similar connectivity requirements. It can span multiple network domains, including the radio access network (RAN) [15, 24, 41], core [46, 68], transport [59] and fronthaul [8]. 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 [13] since the last-mile wireless link is often the bottleneck for mobile apps [4, 6].

Existing approaches [7, 15, 24, 41, 72] 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 §3. 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 [7, 24, 41, 72] 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 state space, consisting of all possible values that the tracked variables can take, grows exponentially in terms of the number of tracked variables. Thus, treating each app as a slice grows the state space exponentially in the number of apps rather than in 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

[1] By "app", we refer to a single mobile application. "User" refers to a mobile device, which can run multiple apps.

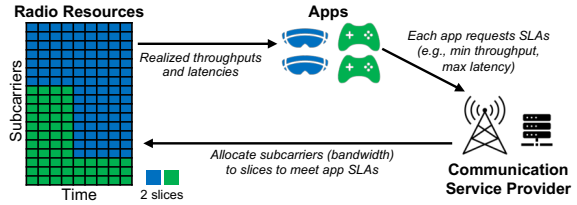


Figure 1: Apps express their connectivity requirements in terms of SLAs, and the operator provisions slice bandwidths to fulfill all SLAs.

slice accommodates tens to hundreds of apps (§3.2).

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 [9, 10, 52, 62] 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 cannot adapt prior approaches [29, 37, 40], which compute required PRBs to support slice-level SLAs, because the state-space complexity precludes treating each app as a slice (§3.3).

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.[2] As illustrated in Fig. 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 bundling apps into slices, and then 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 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 (§4.1).
- 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).
- 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 (§4.3).

We design an O-RAN-compatible [34] architecture to realize these algorithmic concepts (Fig. 4). We have implemented Zipper atop a production-class end-to-end 5G vRAN platform, implementing hooks [25] across different modules in vRAN Distributed Unit (DU) to control slice bandwidth dynamically without compromising real-time performance (§5). 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 (§6); relative to a slice-level service assurance scheduler [41], Zipper reduces SLA violations, measured as a ratio of the violation of the app's request, by 9×.

2 Related Work

RAN slicing. While a static allocation of PRBs to slices provides traffic isolation and simplifies radio resource management [57], it does not guarantee reliable slice performance, since throughput and latency vary with dynamic wireless channel conditions. Orion [24] and SCOPE [7] are slicing-capable RAN virtualization frameworks, and implement existing slice bandwidth schedulers like Network Virtualization Substrate (NVS) [41]. 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 [3, 14, 17, 45, 55].

RadioSaber [15], 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 complementary 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 [72] proposes a reinforcement learning-based

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

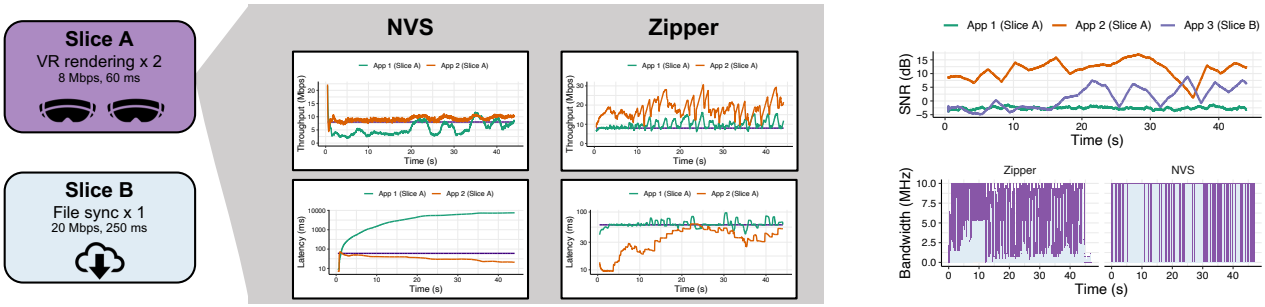


Figure 2: Zipper can efficiently manage an expressive and comprehensive state space to deliver SLAs for each app in each slice.

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.

Admission control. Admission control proposals for RAN slicing cover traffic prediction [58], load balancing [9], pricing [52,62], and game-theoretic formulations [10]. 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 [29,40] are not compatible with Zipper's app-level formulation; we elaborate in §3.3. Recent 5G network slicing proposals [15,24,72] 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 [18,21,65], with a number of proposals across different compute platforms, including CPUs [21,27,61,67], DSPs [5] and GPUs [50]. 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 [30,32]. Canonical algorithms, such as proportionally-fair [63], round-robin, and priority-based [71] schedulers, only consider aggregate throughput and fairness.

3 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 slice bandwidth allocations efficiently.

3.1 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. 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.[3] For example, in Fig. 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 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:

$$\underset{\mathcal{S}_s(t),B_s(t)\,\forall s\in S\,\forall t}{\text{argmin}} \quad \sum_t \sum_{s\in S} B_s(t) \tag{1}$$

$$\text{s.t.} \quad \sum_{s\in S} B_s(t) \leq B \qquad \forall t \tag{2}$$

$$\bar{x}_a(t) \geq x_a^{SLA} \qquad \forall a\in A_s\ \forall s\in S\ \forall t \tag{3}$$

$$\bar{d}_a(t) \leq d_a^{SLA} \qquad \forall a\in A_s\ \forall s\in S\ \forall t, \tag{4}$$

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 Eqn. 3 and Eqn. 4 respectively. Eqn. 2 ensures that the sum of slice bandwidths does not exceed the bandwidth available

[3] 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.

at the base station. The objective in Eqn. 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. 2 visualizes the results achieved by NVS [41], a widely-used slice-level service assurance system [7, 19, 24], 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 Eqn. 1-Eqn. 4 will not have a valid solution. To make the problem tractable, we can relax the constrains in Eqn. 3 and Eqn. 4 into two penalty functions that quantify how far—if at all—an app deviates from its throughput and latency SLAs:

$$f_x^a(t) = \left|\min\left(\bar{x}_a(t) - x_a^{SLA}, 0\right)/x_a^{SLA}\right| \quad (5)$$

$$f_d^a(t) = \left|\min\left(d_a^{SLA} - \bar{d}_a(t), 0\right)/d_a^{SLA}\right| \quad (6)$$

$f_x^a(t)$ in Eqn. 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 (Eqn. 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.

3.2 Challenge: state space complexity

Prior methods [41, 72] monitor aggregate state variables like average slice throughput [41], average channel quality across all users, and average latency across all users [72], to deliver service assurance at the slice level. The search space for such state vectors grows exponentially with the number of slices. Fig. 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 [72] 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.

3.3 Challenge: determining RAN resource availability

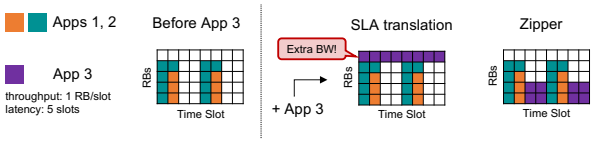


Figure 3: Translating an app's SLAs directly to required slice bandwidth can ignore schedules with greater spectral efficiency.

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 [9, 10, 52, 62], and is not the focus of this paper. However, in order to interface the slice controller with a particular policy, we need a mechanism [52] that answers the following question: *for a pre-determined time period (i.e., 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 [41] to absorb errors that operators make in admitting apps that it cannot support. However, a constant buffer can underutilize the spectrum. Prior work [37] 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 [29, 40] 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. 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

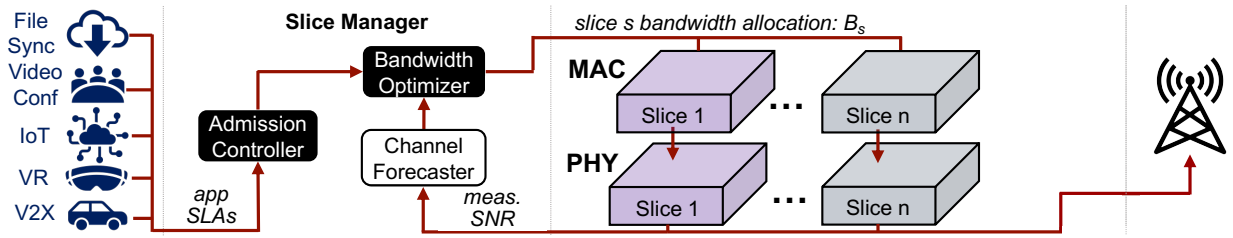


Figure 4: Zipper provisions connectivity by dynamically optimizing network slice bandwidth and resource allocation to meet app-level SLAs.

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 [15, 24] do not address how to interface their slice controllers with operators' admission control policies. Since the RAN is often the bottleneck link [4, 6], it is often oversubscribed. Thus, slicing systems are unusable in practice without a mechanism to estimate resource availability and an accompanying admission control policy.

4 Design

In this section, we describe how we design Zipper, illustrated in Fig. 4 to enable app-level service assurance. Zipper consists of a model predictive control (MPC) framework to manage the search space complexity (§4.1), uses an efficient algorithm to compute slice bandwidth allocations within this MPC formulation (§4.2), and exposes a primitive to help operators forecast RAN resource availability (§4.3).

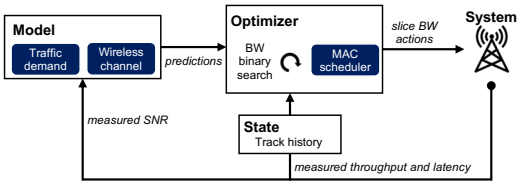


Figure 5: Zipper uses model predictive control (MPC) to compute slice bandwidths that comply with all app SLAs. With MPC, Zipper decouples prediction from control to manage the state space.

4.1 Model predictive control

MPC [26] 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 practical in a number of real-world control problems, including in adaptive bitrate selection for video streaming [66, 70] and in robotics [64].

Fig. 5 illustrates how Zipper applies MPC to solve the optimization problem formulated in Equations 1-4. 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)[4] will evolve over the near term. Forecasting the wireless channel is a well-researched and fundamentally challenging problem [38, 44, 47, 49]. 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.

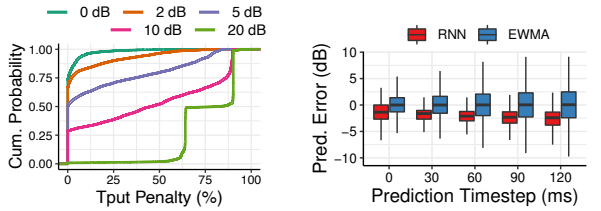
To understand the impact of SNR prediction error on Zipper's ability to meet SLAs, we run a simple experiment,[5] involving a 40 MHz channel and 10 video conferencing apps (i.e., 2 Mbps min throughput, 150 ms max latency), split across 2 slices. We randomly assign each user a 30-second SNR trace gathered 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. 6a visualizes the results as CDFs of the throughput and latency penalties ($f_x^a(t)$ and $f_x^a(t)$ in Eqn. 5 and Eqn. 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 [35], and MAC schedulers use a lookup table to map measured channel quality to MCS [39]. 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. The results from Fig. 6a and this insight about MCS quantization show why MPC is resilient to modeling error in the context of Zipper's scheduling problem.

To forecast each user's channel, we train a sequence-to-

[4] 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 [15] predict SNR at a subcarrier granularity.

[5] We evaluate Zipper more extensively against an Oracle policy in §6.



(a) Resilience to SNR error. (b) SNR prediction error.

Figure 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.

sequence Recurrent Neural Network (RNN) [56], 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. A.1 describes the architecture of this RNN. We train this model using a dataset of SNR traces [54] 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. 6b shows the prediction error (relative to the ground truth) at different points over the 150 ms prediction horizon. Each boxplot 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 finer granularity of 50 ms. Although we use this RNN model in our implementation, Zipper could use any other predictor—including the EWMA filter—with comparable error.

Other state space variables. Zipper tracks the average throughput and latency, $\bar{x}_a(t)$ and $\bar{d}_a(t)$ 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 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 §3.1.

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

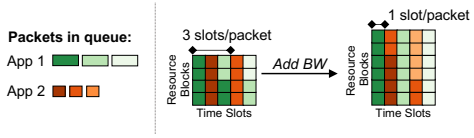


Figure 7: Exposing more bandwidth to a slice reduces packet latency.

apps contending for the same radio resources in the slice.

Monotonicity. Our insight is that both app throughput and app latency are *monotonic functions of slice bandwidth*. Fig. 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. A.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 Eqn. 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 (§4.1). Zipper evaluates the resulting schedule $\tilde{S}_s$ by computing the *penalty scores*, $f_x^a(t)$ and $f_d^a(t)$ as defined in Eqn. 5 and Eqn. 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 $\tilde{S}$ for the allocation $B/2$. For instance, if at least one app's throughput in $\tilde{S}$ 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

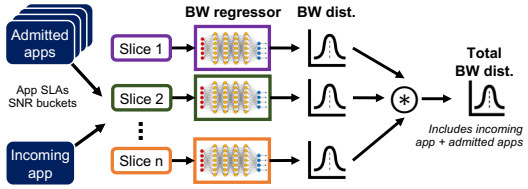


Figure 8: Zipper builds a family of DNNs that forecasts bandwidth distributions for slices consisting of different MAC schedulers and apps with different demand patterns.

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[6] to ensure that Zipper is reactive but not myopic. If $B_s(t)$ violates Eqn. 2 for any slice s, Zipper resolves the conflict to ensure that the allocated bandwidth does not exceed the capacity.

Resolving conflicts. Since Zipper tunes slice bandwidths independently, the allocations could conflict, i.e., $\sum_{s\in S} B_s(t) > B$, which violates Eqn. 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 (§4.3).

App. A.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 [15]. We can also model methods to increase user capacity, such as beam steering [11, 28].

4.3 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 [29, 40] 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 (§3.3). 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. 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 [29, 40] 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 [24]. 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).[7] Zipper drops each incoming app into the best effort slice for a brief period (e.g., 5 seconds), 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.1 describes the DNN architecture.

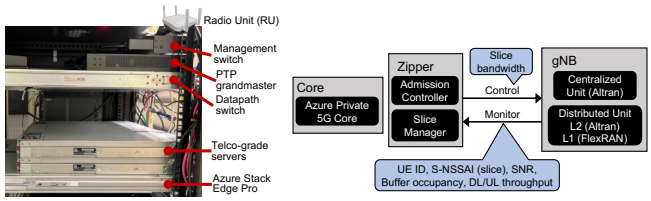
Training DNNs. We generate training data by using Zipper's slice manager as a simulator. We start by enumerating all possible feature embeddings, and run a micro-simulation of Zipper for each embedding using simulated channel traces (assuming a Rayleigh channel model) with SNR values corresponding to 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[8] the slices' independent probability distributions:

[6] 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 (§4.1). A shorter horizon allows us to replan with fresh estimates of SNR.

[7] Note that we only discretize SNR into bins to estimate resource availability with the DNNs; at runtime, the slice controller forecasts SNR (§4.1).

[8] The probability distribution of a sum of independent random variables is



(a) Hardware configuration (b) Software stack

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

$P = P_1 \circledast P_2 \circledast \cdots \circledast P_n$. 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.

5 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. 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 [51], 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 [53]. 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 the hardware devices.

Testbed software. Fig. 9b illustrates the software stack. For the L1, our testbed runs production-ready Intel FlexRAN v20.11 [18]. For the L2/L3, we use Altran 5G vRAN software from Capgemini [2]. A single Telco server hosts both the Altran and FlexRAN software, while Zipper runs on another server. We installed Azure Private 5G Core (AP5GC) [48] 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 1. 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 [34]. 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.

Zipper slice manager. Our implementation of the slice manager (Fig. 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 [43], etc.) from the vRAN via a UDP socket. Zipper populates its state—maintained over a moving horizon T_w (§3.1)—with these measurements. We implemented data buffers to support quick, thread-safe read/write access that meets the stringent slot deadlines for 5G workloads [1].

6 Evaluation

We evaluate Zipper with typical RAN workloads (§6.1) on our production-grade testbed (§6.2) and in emulated environments (§6.3-§6.5). Our evaluation highlights include:

- On our end-to-end testbed, Zipper dynamically tunes slice bandwidths every millisecond to fulfill app SLAs (§6.2).
- Compared to a slice-level service assurance scheduler, Zipper reduces tail throughput and latency penalties as a percentage of app SLAs by 9× (§6.3).
- Zipper can support 150 apps drawn from a typical workload, and incur nearly no penalty in throughput and latency (§6.3).
- 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 (§6.3) .
- Zipper's admission control framework can intelligently allocate unutilized bandwidth, admitting 15% more apps for a first-come-first-served policy (§6.4).
- Zipper supports 200 apps and 70 slices in real time (§6.5).

6.1 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

the convolution of their individual distributions [31].

App Type	Min Tput	Max Latency	QCI [36]	Freq.
Video conf.	2 Mbps	150 ms	40	30%
Voice	200 kbps	100 ms	20	30%
Vehicle-to-X	200 kbps	50 ms	40	10%
Video stream.	2 Mbps	300 ms	60	20%
VR offload	10 Mbps	30 ms	68	5%
File sync	20 Mbps	—	80	5%

Table 1: Apps, SLAs, and frequencies selected for experiments.

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 1 summarizes the SLAs we select for these applications, based on the definitions in the 3GPP specifications [36], 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 [23, 60].

For file sync apps, we instrument iPerf [22] 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 [54]. Our testbed experiment with real client devices (§6.2) 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.[9]

6.2 End-to-end evaluation

We begin by evaluating Zipper end-to-end on our production-grade 5G vRAN testbed (§5), 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.

[9] 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).

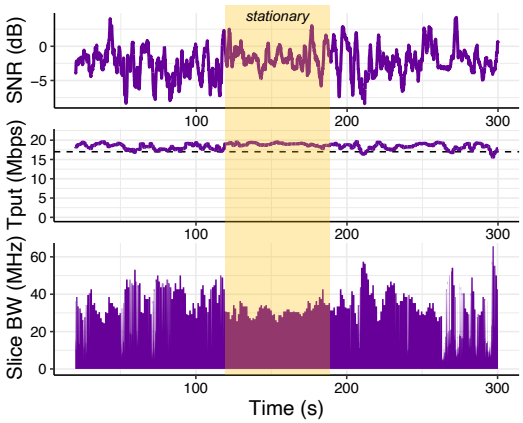


Figure 10: Zipper tunes the bandwidth allocated to a slice serving a mobile OnePlus phone running 17 Mbps iPerf flow.

Fig. 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 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).

6.3 SLA compliance

Setup. In order to evaluate Zipper's ability to comply with SLAs, we use our emulation framework (§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 [63], which is widely used by base stations today [4].
- The *QoS policy* [12, 71], 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 [36]. Table 1 shows the QCIs we use for the apps in our experiments. This policy is also common in production RAN deployments today.
- The *NVS policy* [41] 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

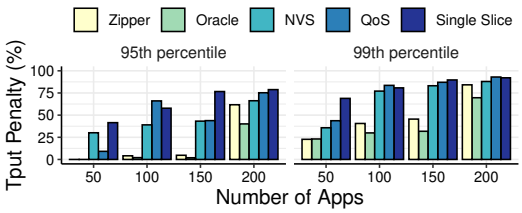


Figure 11: Tail throughput penalties for varying load. Apps scheduled by Zipper experience 95th percentile penalties close to 0%.

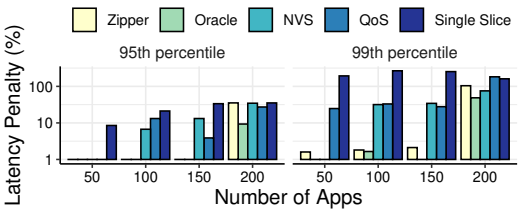


Figure 12: Tail latency penalties for varying base station loads. Apps scheduled by Zipper have low 95th and 99th percentile penalties.

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 [15, 19, 24].

- The *Oracle policy* simply runs Zipper's bandwidth allocation algorithm (§4.2), 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 (§4.1).

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 (Eqn. 5 - Eqn. 6) for each app, after all apps have connected to the 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.[10] 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 [23, 36, 60]. In particular, for each experiment, we compute the throughput and latency penalties, defined in Eqn. 5 - Eqn. 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.

Throughput penalties. Fig. 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 [36], 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. 12 shows the latency penalties (on a log-scale) for the same experiment. The Single 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 (§4.1). The gap widens slightly at 200 apps, since the system is more congested, and thus, assigning a suboptimal MCS would be more consequential.

RAN utilization. Fig. 11 and Fig. 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. 13 shows a 150-second time series snapshot of the aggregate RAN

[10] We assign these flows the lowest QCI priority amongst those in Table 1.

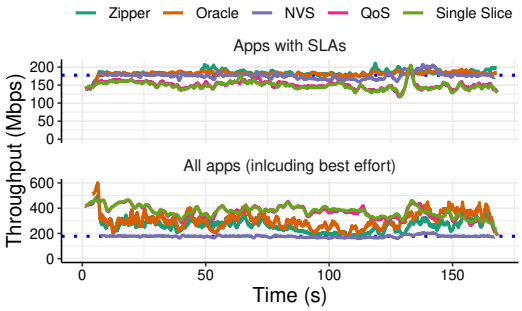
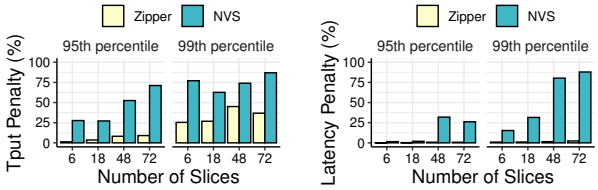


Figure 13: Throughput for 75 apps + 20 best effort apps. Zipper meets the SLAs reliably, and allocates excess capacity to best effort.



(a) Throughput penalty (b) Latency penalty

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

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. 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. 14a and Fig. 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.

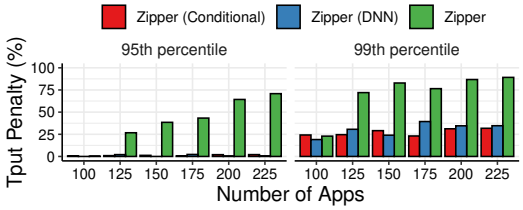


Figure 15: Both the conditional and DNN-based resource estimation methods achieve bounded (and low) penalties.

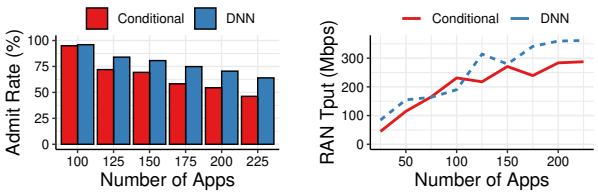
6.4 Forecasting RAN resource availability

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 (§4.3), we admit an incoming app to its designated slice if the p95 bandwidth is within an $\varepsilon = 5$ MHz tolerance of the total bandwidth B, and assigns it to the best effort slice otherwise (i.e., $P < B + \varepsilon$). 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 the best effort slice otherwise. This form of probing resource availability conditionally is common in many admission control proposals [29, 37, 52]. 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 §6.3, we draw apps from the distribution in Table 1, and select app arrival times and contract durations at random.

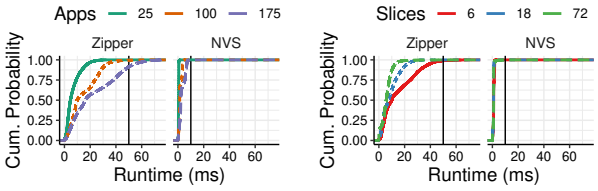
Bounded penalty. A good forecaster of resource availability should ensure that it can satisfy the SLAs of apps that it has already admitted before committing to a new app. This amounts to ensuring that the RAN maintains a bounded and low penalty, as more apps connect to the system. Fig. 15 shows the 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.

Admit rate. While the penalties are bounded, does the RAN have some unutilized capacity that it could have allocated by admitting more apps? Fig. 16a compares the admit rates for



(a) Admit rate (b) RAN utilization

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



(a) Number of apps (b) Number of slices

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

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.

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 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. 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.

6.5 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 1, we profile the time it takes to compute the bandwidth allocations and MAC schedules. We compare Zipper with NVS. Fig. 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).

7 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. 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.

8 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, to accommodate highly mobile users, we can co-optimize slicing across base stations. 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.

Acknowledgments

We thank the reviewers, Mohammad Alizadeh, Hari Balakrishnan, Xenofon Foukas, Anuj Kalia, Radhika Mittal, and Deepak Vasisht for helpful conversations and detailed feedback.

References

[1] A. Al-Dulaimi, X. Wang, and C.-L. I. *Network Slicing for 5G Networks*, pages 327–370. John Wiley & Sons, New Jersey, USA, 2018.

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

[3] S. Bakri, P. A. Frangoudis, A. Ksentini, and M. Bouaziz. Data-driven ran slicing mechanisms for 5g and beyond. *IEEE Transactions on Network and Service Management*, 18(4):4654–4668, 2021.

[4] A. Balasingam, M. Bansal, R. Misra, K. Nagaraj, R. Tandra, S. Katti, and A. 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.

[5] M. Bansal, A. Schulman, and S. 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.

[6] N. Baranasuriya, V. Navda, V. N. Padmanabhan, and S. 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.

[7] L. Bonati, S. D'Oro, S. Basagni, and T. 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.

[8] N. Budhdev, R. Joshi, P. G. Kannan, M. C. Chan, and T. 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.

[9] P. Caballero, A. Banchs, G. de Veciana, and X. Costa-Pérez. Multi-tenant radio access network slicing: Statistical multiplexing of spatial loads. *IEEE/ACM Transactions on Networking*, 25(5):3044–3058, 2017.

[10] P. Caballero, A. Banchs, G. de Veciana, X. Costa-Pérez, and A. Azcorra. Network slicing for guaranteed rate services: Admission control and resource allocation games. *IEEE Transactions on Wireless Communications*, 17(10):6419–6432, 2018.

[11] Z. Cao, Q. Ma, A. B. Smolders, Y. Jiao, M. J. Wale, C. W. Oh, H. Wu, and A. M. J. 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.

[12] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda. Downlink packet scheduling in lte cellular networks: Key design issues and a survey. *IEEE communications surveys & tutorials*, 15(2):678–700, 2012.

[13] M. Chahbar, G. Diaz, A. Dandoush, C. Cérin, and K. Ghoumid. A comprehensive survey on the e2e 5g network slicing model. *IEEE Transactions on Network and Service Management*, 18(1):49–62, 2020.

[14] C.-Y. Chang and N. Nikaein. Ran runtime slicing system for flexible and dynamic service execution environment. *IEEE Access*, 6:34018–34042, 2018.

[15] Y. Chen, R. Yao, H. Hassanieh, and R. Mittal. Channel-aware 5g ran slicing with customizable schedulers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[16] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[17] E. Coronado and R. 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.

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

[19] X. Costa-Pérez, J. Swetina, T. Guo, R. Mahindra, and S. Rangarajan. Radio access network virtualization for future mobile carrier networks. *IEEE Communications Magazine*, 51(7):27–35, 2013.

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

[21] J. Ding, R. Doost-Mohammady, A. Kalia, and L. Zhong. *Agora: Real-Time Massive MIMO Baseband Processing in Software*, page 232–244. Association for Computing Machinery, New York, NY, USA, 2020.

[22] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. Iperf. https://iperf.fr/, 2022.

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

[24] X. Foukas, M. K. Marina, and K. 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.

[25] X. Foukas, B. Radunovic, M. Balkwill, and Z. Lai. Taking 5g ran analytics and control to a new level. In *Technical Report*, December 2022.

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

[27] K. C. Garikipati, K. Fawaz, and K. 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.

[28] Y. Ghasempour, M. K. Haider, and E. W. Knightly. Decoupling beam steering and user selection for mu-mimo 60-ghz wlans. *IEEE/ACM Transactions on Networking*, 26(5):2390–2403, 2018.

[29] T. Guo and A. Suárez. Enabling 5g ran slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[30] T. Guo and A. Suárez. Enabling 5g ran slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

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

[32] Y. Huang, S. Li, Y. T. Hou, and W. 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.

[33] M. Hüsken and P. Stagge. Recurrent neural networks for time series classification. *Neurocomputing*, 50:223–235, 2003.

[34] C.-L. I and S. Katti. O-ran: Towards an open and smart ran. Technical report, Open RAN Alliance, 2018.

[35] E. T. S. Institute. *Physical layer procedures for data*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[36] E. T. S. Institute. *System Architecture for the 5G System*. ETSI 3rd Generation Partnership Project (3GPP), 06 2018.

[37] M. Jiang, M. Condoluci, and T. Mahmoodi. Network slicing management & prioritization in 5g mobile systems. In *European Wireless 2016; 22th European Wireless Conference*, pages 1–6, 2016.

[38] W. Jiang and H. D. Schotten. Deep learning for fading channel prediction. *IEEE Open Journal of the Communications Society*, 1:320–332, 2020.

[39] M. T. Kawser, N. I. B. Hamid, M. N. Hasan, M. S. Alam, and M. M. Rahman. Downlink snr to cqi mapping for different multipleantenna techniques in lte. *International journal of information and electronics engineering*, 2(5):757, 2012.

[40] B. Khodapanah, A. Awada, I. Viering, D. Oehmann, M. Simsek, and G. 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.

[41] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. Nvs: A substrate for virtualizing wireless resources in cellular networks. *IEEE/ACM Transactions on Networking*, 20(5):1333–1346, 2012.

[42] S. Kumar, E. Hamed, D. Katabi, and L. Erran Li. Lte radio analytics made easy and accessible. *SIGCOMM Comput. Commun. Rev.*, 44(4):211–222, aug 2014.

[43] H. Lee, S. Noghabi, B. Noble, M. Furlong, and L. Cox. Bumblebee: Application-aware adaptation for edge-cloud orchestration. In *Symposium on Edge Computing*. ACM/IEEE, December 2022.

[44] J. Lee, S. Lee, J. Lee, S. D. Sathyanarayana, H. Lim, J. Lee, X. Zhu, S. Ramakrishnan, D. Grunwald, K. Lee, and S. 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.

[45] J. Li, W. Shi, P. Yang, Q. Ye, X. S. Shen, X. Li, and J. Rao. A hierarchical soft ran slicing framework for differentiated service provisioning. *IEEE Wireless Communications*, 27(6):90–97, 2020.

[46] X. Li, C. Guo, L. Gupta, and R. Jain. Efficient and secure 5g core network slice provisioning based on vikor approach. *IEEE Access*, 7:150517–150529, 2019.

[47] C. Luo, J. Ji, Q. Wang, X. Chen, and P. 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.

[48] Microsoft. Azure private 5g core. Technical report, Microsoft, 2023.

[49] L. S. Muppirisetty, T. Svensson, and H. Wymeersch. Spatial wireless channel prediction under location uncertainty. *IEEE Transactions on Wireless Communications*, 15(2):1031–1044, 2015.

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

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

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

[53] Qulsar. Qulsar qg2. https://qulsar.com/Products/Systems/Qg_2.html, 2022.

[54] D. Raca, D. Leahy, C. J. Sreenan, and J. 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.

[55] D. A. Ravi, V. K. Shah, C. Li, Y. T. Hou, and J. H. Reed. Ran slicing in multi-mvno environment under dynamic channel conditions. *IEEE Internet of Things Journal*, 9(6):4748–4757, 2022.

[56] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

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

[58] V. Sciancalepore, K. Samdanis, X. P. Costa, D. Bega, M. Gramaglia, and A. 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.

[59] N. Shahriar, S. Taeb, S. R. Chowdhury, M. Zulfiqar, M. Tornatore, R. Boutaba, J. Mitra, and M. 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.

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

[61] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, jan 2011.

[62] S. Troia, A. F. R. Vanegas, L. M. M. Zorello, and G. Maier. Admission control and virtual network embedding in 5g networks: A deep reinforcement-learning approach. *IEEE Access*, 10:15860–15875, 2022.

[63] D. Tse and P. Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, Cambridge, UK, 2005.

[64] P. 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.

[65] J. Wu, Z. Zhang, Y. Hong, and Y. Wen. Cloud radio access network (c-ran): a primer. *IEEE network*, 29(1):35–41, 2015.

[66] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. 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.

[67] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang. Bigstation: Enabling scalable real-time signal processingin large mu-mimo systems. *SIGCOMM Comput. Commun. Rev.*, 43(4):399–410, aug 2013.

[68] Q. Ye, J. Li, K. Qu, W. Zhuang, X. S. Shen, and X. 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.

[69] W. Yin, K. Kann, M. Yu, and H. Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.

[70] X. Yin, A. Jindal, V. Sekar, and B. 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.

[71] Y. Zaki, T. Weerawardane, C. Görg, and A. 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.

[72] L. Zanzi, V. Sciancalepore, A. Garcia-Saavedra, H. D. Schotten, and X. Costa-Pérez. Laco: A latency-driven network slicing orchestration in beyond-5g networks. *IEEE Transactions on Wireless Communications*, 20(1):667–682, 2021.

A Allocating Slice Bandwidth in Zipper

A.1 Forecasting the wireless channel with an RNN

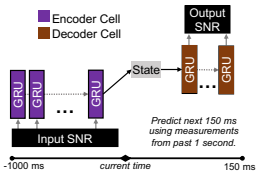


Figure 18: Architecture of RNN model to forecast wireless channel.

To forecast each user's channel, we train a sequence-to-sequence Recurrent Neural Network (RNN) [56], 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) [16], 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 [33] and natural language models [69]. 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. 18 illustrates the architecture of this RNN.

RNNs are emerging a popular method to forecast timeseries, including wireless channel [38, 44, 47, 49]. We develop and train a model, but note that Zipper can support any predictor of wireless channel. §4.1 characterizes the requirements for suitable predictor.

A.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 [4, 42]. 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 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.

A.3 Algorithm

Algorithm 1 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.

Algorithm 1 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 Σ_{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)
```

B Estimating resource availability in Zipper

B.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. 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.