TOWARDS HIGH-ACCURACY AND RESOURCE-EFFICIENT EDGE-ASSISTED AUGMENTED REALITY

by

Qiang Xu

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Y. Charlie Hu, Chair

School of Electrical and Computer Engineering

Dr. Samuel P. Midkiff

School of Electrical and Computer Engineering

Dr. James C. Davis

School of Electrical and Computer Engineering

Dr. Lu Su

School of Electrical and Computer Engineering

Approved by:

Dr. Milind Kulkarni

To my parents and Lingyu

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude towards my advisor, Prof. Y. Charlie Hu. His insights ignited a spark of curiosity within me, pushing me beyond my comfort zone to explore uncharted territories in research. The journey wasn't always smooth sailing — there were moments of doubt and setbacks. But through his unwavering support, we navigated those challenges together. His mentorship instilled in me a deep sense of ownership for my work, a valuable lesson that will forever guide my progress and success.

I would also like to express my appreciation towards my advisory committee members, Prof. Samual P. Midkiff, Prof. James C. Davis, and Prof. Lu Su. Their insightful advice and thoughtful critiques were instrumental in shaping my research and propelling it forward.

Furthermore, I extend my gratitude to my exceptional lab mates. The DSNL lab wasn't just a workplace; it became my second family. Dr. Akshay Jajoo, Dr. Sibendu Paul, Dr. Jiayi Meng, Pranab Dash, Z. Jonny Kong, Omar Basit, and Zongpu Zhang — you all played a pivotal role in making my research journey enriching and unforgettable. The camaraderie we shared, the collaborative spirit that fueled our discussions, and the constant exchange of ideas not only fostered groundbreaking research but also forged deep bonds of friendship that I will forever cherish.

I am also immensely grateful to my collaborators, Prof. Dimitrios Koutsonikolas, Dr. Shivang Aggarwal, Moinak Ghoshal, Imran Khan, Phuc Dinh, and Zixiao Lu from Northeastern University, Prof. Abhilash Jindal from IIT Delhi, Prof. Yuanjie Li from Tsinghua University, and Dr. Srimat T. Chakradhar, Murugan Sankaradas, and Ravi Kailasam Rajendran from NEC Laboratories America, Inc. Their willingness to share their knowledge wasn't just helpful, it sparked new ideas and fostered a truly collaborative environment. The fruitful discussions we shared, where diverse perspectives were openly exchanged, expanded the scope of my research and pushed it in exciting new directions.

Words cannot express my gratitude to my parents, Mr. Jutang Xu and Mrs. Ping Yan. Their unwavering love and support have been the bedrock of my achievements. They not only encouraged me to chase my research dreams but also nurtured the values that shaped me into the person I am today.

Last but not least, I want to thank my partner, Lingyu Ma, for her continuous patience and enduring support. We've celebrated every victory, big and small, and weathered every storm together. Your steadfast faith in me has been a constant source of strength, pushing me forward even when doubt threatened to cloud my vision. I cherish every moment and memory we've built, and I look forward to many more. A special acknowledgment to my furry friend, Anan. You may not have understood the complexities of my research, but your presence was a constant reminder of the simple things in life that bring joy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Immersive applications such as augmented reality (AR) and mixed reality (MR) often need to perform latency-critical analytics tasks on every frame captured on camera. These tasks, often powered by deep neural networks (DNNs) for their superior accuracy, necessitate offloading to edge servers with GPUs due to their computational intensity. Achieving high accuracy and efficient AR task offloading faces two fundamental challenges untapped by prior work: (1) In practice, multiple DNN-supported tasks need to offload concurrently to achieve the app functionality — how to schedule such offloaded tasks on the client which compete for shared edge server resources to maximize the app QoE? (2) Concurrent AR clients from a large user base offload to a cluster of GPU servers — how to schedule the offloaded tasks on the servers to maximize the number of clients served and lower the operating cost?

To tackle the first challenge, we design a framework, AccuMO, that balances the offloading frequencies of different tasks by dynamically scheduling the offloading of multiple tasks from an AR client to an edge server, thereby optimizing the overall accuracy across tasks and hence app QoE. Our design employs two novel ideas: (1) task-specific lightweight models that predict offloading accuracy drop as a function of offloading frequency and frame content, and (2) a general two-level control feedback loop that concurrently balances offloading among tasks and adapts between offloading and using local algorithms for each task.

We tackle the challenge of supporting concurrent AR clients in two steps. We first focus on maximizing the capacity of individual edge servers, where we present ARISE, which untangles the intricate interplay between per-client offloading schedule and batched inference on the server by proactively coordinating offloading requests from different AR clients. In the second step, we focus on a cluster setup of heterogeneous GPU servers which exposes the synergy between diversity in both DNN layers and GPU architectures, manifesting as comparable inference latency for many layers in DNN models when running on low-class and high-class GPUs. We exploit such overlooked capability of low-class GPUs using pipeline parallelism and present a novel inference serving system, IPIPE, that employs pool-based pipeline parallelism with a mixed-integer linear programming (MILP)-based control plane and a data plane that performs resource reservation-based adaptive batching.

1. INTRODUCTION

1.1 Edge-Assisted Augmented Reality

Driven by the expanding applications and technological advancements, Augmented Reality (AR) is experiencing significant growth. The global AR market is projected to reach \$88.4 billion by 2026, growing at a compound annual growth rate (CAGR) of 31.5% [1]. Meanwhile, AR is expending into various sectors, including entertainment, healthcare, retail, automotive, and education, offering innovative solutions like augmented surgeries, interactive driving experiences, and immersive e-learning. This widespread adoption underscores AR's potential to revolutionize multiple industries, making it a pivotal technology for the future.

Immersive AR applications often need to perform a number of challenging tasks to provide enhanced user experience. Consider Pokemon Go, a representative AR app, where players use their mobile devices to capture, train and battle with virtual creatures called Pokemon which appear as if they exist in the real-world environment. Supporting realistic, interactive, and immersive user experience such as allowing Pokemon to hide behind a tree or jump on a tree branch requires performing several essential computer vision tasks for each frame, at the high frame rate of the camera capture, *e.g.,* every 16.7 ms: (1) **Odometry**: While a player moves within her real-world surroundings, the mobile device needs to track the camera pose to render virtual objects (*e.g.,* Pokemon) at the correctly perceived locations; (2) **Depth estimation**: The mobile device needs to process the distance information between the camera and physical objects, in order to correctly render virtual objects into the physical environment, *e.g.,* allowing Pokemon to hide behind a tree; (3) **Object detection**: Being able to identify physical objects in each camera frame enhances the virtual objects with spatial and contextual awareness of their surrounding physical world and allows them to interact accordingly, *e.g.,* if a tree branch is nearby, the Pokemon jumps onto it.

Deep Neural Network (DNN) models have been increasingly used to support these complex AR tasks due to their high accuracy (*e.g.,* [2]–[7]). In contrast, traditional AR frameworks, *e.g.,* ARCore, have several known limitations including low resolution and only getting accurate results for up to 5 meters for depth estimation [8]. However, the high-accuracy, DNN-based solutions are generally computationally heavy. Running state-of-the-art DNN models on commodity mobile devices could take hundreds of milliseconds or even seconds [9]–[11]. To attain high accuracy results on resource-constrained devices, and propelled by the availability of faster wireless networks such as 5G [12], offloading (also known as edge-assisted solutions) has been proposed [9], [13], [14], where camera frames are uploaded to a cloud or edge server for DNN inference.

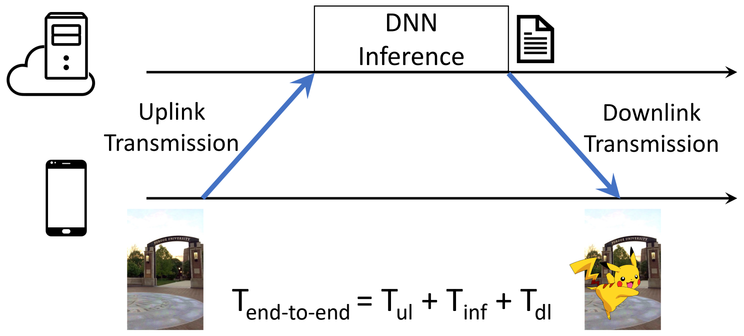


Figure 1.1. Workflow of edge-assisted AR.

As shown in Figure 1.1, in edge-assisted AR, the AR client continuously upload its camera frames to the edge server via a fast wireless network such as 5G. The edge server, which often resides in the same city and is connected to the 5G infrastructure via low-latency links [15], runs the high-accuracy DNN model on its GPU and sends the inference results back via the downlink. Upon receiving these results, the AR client performs higher-level tasks, *e.g.*, rendering, that rely on the inference results. The end-to-end offloading latency comprises the frame transfer latency, the inference latency, and the result transfer latency.

1.2 Challenges in Edge-Assisted AR Deployment

The large amount of recent work on edge-assisted AR have focused on offloading a single task at a time, in particular, object detection (*e.g.*, [16]–[19]), for a single user (AR app), who is allocated a dedicated edge server or GPU. However, deploying practical AR apps surpasses this simplistic setup in two important ways. Firstly, as discussed in §1.1, multiple DNN-supported tasks need to be offloaded concurrently to achieve the app functionality. Secondly, a successful commercial AR app often needs to support a large user base. Given the increased workload due to the two factors, while the simplistic setup can potentially be extended to offloading multiple tasks and supporting multiple users by allocating each offloaded task and each user a dedicated GPU, such an approach is not cost-effective and not scalable. In fact, to control the server cost, in a shared edge cloud, a user may only be allocated a slice of a GPU [20]–[22]. For example, in Amazon Elastic Inference, GPUs are partitioned and priced per TFLOPS [20].

To bridge the gap between existing edge-assisted AR setup and the requirement for practical edge-assisted AR app deployment, we face two major challenges:

Challenge 1: How to schedule multiple offloaded tasks on the client to maximize the app QoE? Offloading multiple tasks of a latency-critical app faces a new design objective beyond that for offloading a single task. In offloading a single task, when the end-to-end offloading latency is longer than a frame interval, which happens often due to server inference and frame transfer delay, the current frame accuracy suffers from staleness of the last server-returned result. Hence, in offloading a single task, the primary goal is to reduce the end-to-end offloading latency, as the lower the end-to-end offloading latency, the less stale the last server-returned result, and the higher the task accuracy. In contrast, when an app needs to offload multiple tasks, which compete for shared resources allocated to a user (*e.g.,* edge server GPU), offloading of different tasks have to be interleaved in some manner, which increases the staleness of the last returned server inference result for each task and adversely affects the accuracy of all tasks. Since the accuracy of different tasks can affect the app QoE differently, in addition to reducing the end-to-end offloading latency for each task, multitask offloading faces a new design goal: *how to balance the offloading of different tasks to maximize the overall accuracy that ultimately determines the app QoE.*

Challenge 2: How to schedule the execution of offloaded tasks on the servers to maximize the number of clients served? To actually deploy AR apps based on edge-assisted AR design requires effective edge server support for serving potentially a large number of concurrent offloading requests of AR tasks from many mobile clients. Instead of maintaining its own servers, a cost-effective way is for an AR app vendor to use (or build) Machine-Learning-as-a-Service (MLaaS) [23], [24], *e.g.,* deployed in the edge cloud [15], to serve the AR clients in the vicinity. A primary business objective of an MLaaS provider is to increase the capacity of the cluster (*i.e.,* the number of AR clients that it can serve concurrently), or provision the cluster size for a given user base in order to maximize its cost-effectiveness. In such a shared edge-cloud setting, the AR apps running on the mobile clients often exhibit diversity in task accuracy requirements. For example, healthcare apps [25] require high accuracy while tourism apps can get by with moderate accuracy [26]. In such a deployment, the vendor or the users of the AR apps specify an accuracy SLA for each AR task (*e.g.,* object detection) that is required to ensure satisfactory app QoE, and the MLaaS provider tries to maximize the capacity of the GPU cluster in concurrently serving multiple AR clients. We formally state the *AR offloading inference serving* problem as: *Given an edge/cloud server cluster serving offloaded inference tasks from AR clients, how to maximize the number of clients supported by individual servers while meeting the per-client AR task accuracy SLAs?*

1.3 Thesis Contributions

Thesis statement. *As edge-assisted AR workloads scale with increasing numbers of tasks, clients, and servers, effective scheduling is critical to achieve high accuracy and resource efficiency. The unique characteristics of AR workloads present new optimization opportunities, enabling the design of more effective schedulers.*

This thesis makes several contributions in addressing the two key challenges impeding the deployment of edge-assisted AR. To tackle the first challenge, we design a framework that balances the offloading frequencies of different tasks by dynamically scheduling the offloading of multiple DNN tasks from an AR client to an edge server, thereby optimizing the overall accuracy across tasks and hence app QoE. We tackle the challenge of supporting concurrent AR clients in two steps. We first focus on maximizing the capacity of individual edge servers, where we exploit the intricate interplay between per-client offloading schedule and batched inference on the server via proactively coordinating offloading request streams from different AR clients. In the second step, we focus on a cluster setup of heterogeneous GPU servers which exposes the synergy between diversity in DNN layers and diversity in

GPU architectures, and we propose pool-based pipeline parallelism to significantly improve the utilization of low-class GPUs.[1]

Key Contribution 1: Optimizing AR app accuracy by dynamically scheduling the offloading of multiple DNN tasks. We design a framework called AccuMO that dynamically schedules offloading of multiple compute-intensive DNN tasks of an AR app from a mobile device to an edge server while optimizing the overall accuracy across the tasks. Our design is motivated by two key insights we made about AR tasks: (1) offloading accuracy losses are content-dependent, and different tasks may be affected by different content features; (2) although offloading solutions achieve higher accuracies than directly running a local algorithm on the device, *e.g.,* a lite model, on average, the local algorithm may perform better on selected frames, *e.g.,* when the camera is moving fast. In addition to maximizing the overall accuracy of the tasks, an additional design goal of the AccuMO framework is to easily support any mix of app tasks. To achieve this, we develop a modular design that runs two concurrent control loops that both adapt according to frame content dynamics: a low-level control loop is per task and adapts between local tracking and using the local algorithm for each frame, and a global control loop runs model predictive control (MPC) [27] to dynamically balance offloading of the tasks. We demonstrate that our framework improves the overall accuracy significantly in jointly offloading two core tasks in AR — depth estimation and odometry — by on average 7.6%–14.3% over the best baselines.

Key Contribution 2: Optimizing individual edge server capacity via coordination among AR clients. We present an AR offloading inference framework ARISE which untangles the intricate interplay among the control knobs across clients via a centralized but scalable scheduler that *proactively* coordinates the offloading schedules of AR clients and the batched inference on the edge server. We first develop a novel lightweight, online accuracy estimator that estimates the AR task accuracy for the current frame for each AR client under different offloading frequency, E2E latency, and dynamically changing frame content. We then design a novel scheduler that decouples deriving per-client offloading schedules and server batching schedule in two steps: (1) it first calculates a pseudo-optimal offload-

[1]↑The series of work was done jointly with Z. Jonny Kong with equal contributions.

ing frequency per-client leveraging the accuracy estimator; (2) it then greedily packs future offloaded requests from all clients into fewer large batches and coordinates client requests accordingly without violating per-client accuracy requirements. Our evaluation using a large set of emulated AR clients and a 10-phone testbed shows that ARISE supports 1.7x–6.9x more AR clients compared to various baselines while keeping the per-client accuracy within the client-specified accuracy SLAs.

Key Contribution 3: Optimizing the throughput of heterogeneous GPU clusters with pipeline parallelism. We design IPIPE, a model serving system that harnesses mixed GPU types in heterogeneous GPU clusters via pipelined model inference to maximize its serving throughput. IPIPE is built on three key ideas. First, to support maximal scheduling flexibility, it employs *pool-based pipeline parallelism* where each model partition is associated with a *pool* of GPU servers, and each request can be processed by any GPU allocated to each partition pool along the pipeline. Second, to realize the scheduling flexibility exposed by pool-based pipelined model inference, IPIPE generates the optimal configuration of pool-based pipelined model inference, *e.g.,* one that maximizes the model serving throughput of a given cluster while meeting inference SLOs, using Mixed Integer Linear Programming (MILP), which works as the control plane of IPIPE. Third, to bridge the gap between MILP solution and runtime dynamics due to asynchronous and bursty request arrivals, we employs a separate data plane that performs *resource reservation-based adaptive batching*, which overcomes a major limitation of adaptive batching used in previous work on pipelined inference serving [28]. Evaluation results on diverse workloads (18 CNN models) show that IPIPE achieves at least 58.6% higher utilization of low-class GPUs while maintaining high utilization of high-class GPUs, leading to 15.9%–64.0% higher serving throughput compared to various baselines.

2. AccuMO: ACCURACY-CENTRIC MULTITASK OFFLOADING IN EDGE-ASSISTED MOBILE AUGMENTED REALITY

This chapter is based on work published in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking* [29], https://doi.org/10.1145/3570361.3592531.

2.1 Introduction

Fueled by the rise of metaverse, immersive mobile apps such as Augmented Reality (AR) often need to perform a number of challenging tasks to provide enhanced user experience. Consider Pokemon Go, a popular AR app, where players use their mobile devices to capture, train and battle with virtual creatures called Pokemon which appear as if they exist in the real-world environment. Supporting realistic, interactive, and immersive user experience such as allowing Pokemon to hide behind a tree or jump on a tree branch requires performing several essential computer vision tasks for each frame, at the high frame rate of the camera capture, *e.g.,* every 16.7 ms: (1) **Odometry**: While a player moves within her real-world surroundings, the mobile device needs to track the camera pose to render virtual objects (*e.g.,* Pokemon) at the correctly perceived locations; (2) **Depth estimation**: The mobile device needs to process the distance information between the camera and physical objects, in order to correctly render virtual objects into the physical environment, *e.g.,* allowing Pokemon to hide behind a tree; (3) **Object detection**: Being able to identify physical objects in each camera frame enhances the virtual objects with spatial and contextual awareness of their surrounding physical world and allows them to interact accordingly, *e.g.,* if a tree branch is nearby, the Pokemon jumps onto it.

Equally importantly, all of the tasks of such an AR app performed on each frame are latency-critical; the results for the current frame need to be available in the *current frame interval*, as otherwise they will miss the rendering for the current frame, as dictated by the stringent QoE of AR apps, *e.g.,* 60 FPS [9]. We note this is a critical difference from latency-sensitive applications such as video analytics pipelines where frames of surveillance cameras are uploaded to the cloud for real-time analytics which can tolerate a delay of hundreds of milliseconds [30]. To clearly distinguish the two scenarios, we denote the accuracy returned by AR tasks as the *current-frame accuracy* (CFA), and that for video analytics pipelines as *delayed-frame accuracy* (DFA).

Deep Neural Network (DNN) models have been increasingly used to support these complex AR tasks due to their high accuracy (*e.g.,* [2]–[7]). In contrast, traditional AR frameworks, *e.g.,* ARCore, have several known limitations including low resolution and only getting accurate results for up to 5 meters for depth estimation [8]. However, the high-accuracy, DNN-based solutions are generally computationally heavy. Running state-of-the-art DNN models on commodity mobile devices could take hundreds of milliseconds or even seconds [9]–[11]. To attain high accuracy results on resource-constrained devices, offloading (also known as edge-assisted solutions) has been proposed [9], [13], [14], where camera frames are uploaded to a cloud or edge server for DNN inference.

Despite the importance of offloading multiple tasks of an AR app, the large amount of recent work on edge-assisted AR have focused on offloading a single task at a time, in particular, object detection (*e.g.,* [16]–[19]), assuming a user (AR app) is allocated a dedicated edge server or GPU. Such solutions can potentially be applied to offloading multiple tasks of an AR app of a user by allocating each offloaded task a dedicated GPU. However, such an approach is not cost-effective and not scalable. In fact, to control the server cost, in a shared edge cloud, a user may only be allocated a slice of a GPU [20]–[22]. For example, in Amazon Elastic Inference, GPUs are partitioned and priced per TFLOPS [20].

Offloading multiple tasks of a latency-critical app faces a new design objective beyond that for offloading a single task. In offloading a single task, when the end-to-end offloading latency is longer than a frame interval, which happens often due to server inference and frame transfer delay, the task result for the current frame is generated by either directly returning the last server-returned result (*e.g.,* [10]) or by applying some local tracking technique to that result (*e.g.,* [9], [10], [16], [31]). In both cases, the current frame accuracy suffers from staleness of the last server-returned result. Hence, in offloading a single task, the primary goal is to reduce the end-to-end offloading latency, as the lower the end-to-end offloading latency, the less stale the last server-returned result, and the higher the task accuracy.

In contrast, when an app needs to offload multiple tasks, which compete for shared resources allocated to a user (*e.g.,* edge server GPU), offloading of different tasks have to be interleaved in some manner which increases the staleness of the last returned server inference result for each task and adversely affects the accuracy of all tasks. Since the accuracy of different tasks can affect the app QoE differently, in addition to reducing the end-to-end offloading latency for each task, multitask offloading faces a new design goal: *how to balance the offloading of different tasks to maximize the overall accuracy that ultimately determines the app QoE.* Existing works on multi-DNN inference scheduling (*e.g.,* [32]–[36]) only focus on maximizing throughput or reducing SLO violation, without consideration for accuracy.

In general, the combination of task accuracies that optimize the app QoE is app specific and is beyond the scope of this chapter. We assume such a function $accuracy\_merge()$ that merges the accuracies of all app tasks into a single *overall accuracy* metric is given, *e.g.,* by the app developer. We formally state the **accuracy-centric multitask offloading (AccuMO)** problem:

Given the function $accuracy\_merge()$ *for an app that offloads multiple tasks* $t_1, \ldots, t_k$ *and the server resource constraints, how to schedule the offloading of the tasks to maximize the overall accuracy* $accuracy\_merge(t_1, \ldots, t_k)$*?*

Tackling the above multitask offloading scheduling problem faces two challenges: (1) How to estimate the accuracy or accuracy drop if a task is offloaded under a candidate schedule? (2) How should the scheduler balance offloading of different tasks in an online manner given the interdependence between current and future offloading decisions, while maximizing the overall accuracy?

In this chapter, we present a framework called AccuMO that dynamically schedules offloading of multiple compute-intensive DNN tasks of an AR app from a mobile device to an edge server while optimizing the overall accuracy across the tasks. Our design is motivated by two key insights we made about AR tasks: (1) local tracking accuracy drop rates are content-dependent, and different tasks may be affected by different content features; (2) although offloading solutions achieve higher accuracies than directly running a local algorithm on the device, *e.g.,* a lite model, on average, the local algorithm may perform better on selected frames, *e.g.,* when the camera is moving fast and local tracking suffers from low continuity across consecutive frames.

In addition to maximizing the overall accuracy of the tasks, an additional design goal of the AccuMO framework is to easily support any mix of app tasks. To achieve this, we develop a modular design that runs two concurrent control loops that both adapt according to frame content dynamics: a low-level control loop is per task and adapts between local tracking and using the local algorithm for each frame, and a global control loop runs model predictive control (MPC) [27] to dynamically balance offloading of the tasks.

We have implemented the AccuMO framework and two core tasks of immersive AR apps — depth estimation and odometry — on commodity Android phones and GPU servers. Our evaluation using a large set of videos with diverse frame content and camera movement show that in jointly offloading the two core tasks in AR on commodity devices, AccuMO improves the overall task accuracy over the best baseline by 2.3%–11.9% (avg. 7.6%), 6.7%–14.6% (avg. 10.1%), 10.8%–16.7% (avg. 14.3%), 2.3%–15.2% (avg. 11.2%), and -2.8%–23.9% (avg. 11.2%) under 5 diverse accuracy weight ratios. In particular, AccuMO improves the accuracies of both tasks simultaneously over round-robin scheduling by up to 13.4% for depth estimation and up to 33.8% for odometry.

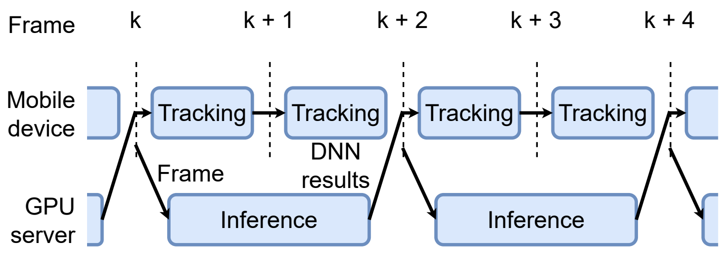
In summary, our main contributions are as follows:

- We present, to our best knowledge, the first accuracy-centric multitask offloading framework that jointly optimizes the overall accuracy of multiple tasks of an AR app running on a mobile device to an edge server.
- We present a novel two-level control feedback loop design that allows for easily adding new tasks while optimizing the overall accuracy across the tasks.
- We also present the first, complete edge-assisted offloading design for two core AR tasks, depth estimation and odometry, which includes local trackers, local algorithms, and novel accuracy models, which are used to dynamically select between them.
- We implement and experimentally validate our AccuMO framework design by comparing it with various static offloading schemes.

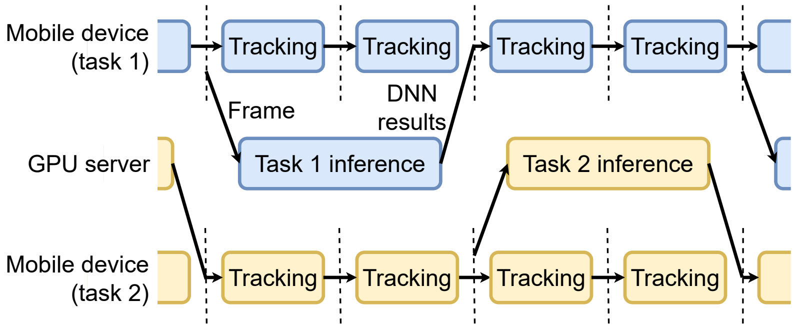
2.2 Background: The Offloading + Local Tracking Paradigm

In this section, we give a brief background on the popular offloading + local tracking paradigm.

In edge-assisted AR, even with powerful GPUs, a typical DNN inference still takes tens of milliseconds, failing to return the result within the same frame interval. For example, models in Meta's object detection model zoo [37] have a median inference time of 52.5 ms on Tesla V100, much longer than the per-frame time of 16.7 ms needed by an AR app running at 60 FPS [9]. In such cases, the result of an offloaded frame may come back several frame intervals later, and stale server-returned results have to be used for the interim frames, resulting in reduced accuracy.



(a) Offloading one task to the GPU server.



(b) Offloading two tasks to the GPU server, the average tracking stride increases from 2.5 to 3.5.

Figure 2.1. The offloading + local tracking paradigm.

To mitigate the staleness of server DNN inference results, *local tracking* has been proposed to generate more accurate task results for the current frames than simply using the last returned result from the server [9], [10], [16], [26], [31], [38]–[45]. Specially, a local tracker runs on the mobile device and adjusts the DNN inference results for the last offloaded frame

f_l sent back by the server to generate refined results for the current frame f_c, by analyzing the changes between the stale frame f_l and the current frame f_c, as shown in Figure 3.1. We denote edge-assisted solutions that exploit local tracking as the *offloading + local tracking paradigm*. Local trackers are fast and can typically finish within one frame time. They are also task-specific and often custom-designed for each type of tasks. For example, for object detection, local trackers use motion vectors to estimate the movement of an object within a bounding box, and adjust the bounding box accordingly [9], [44].

While local tracking improves the accuracy of the result for the current frame f_c (compared to directly reusing the last server returned result), the gap between its accuracy and that of running the server DNN model (if we could), denoted as *accuracy drop*, still widens with *tracking stride*, defined as the frame distance between f_l and f_c, due to increased staleness of the results for frame f_l. For example, in Figure 3.1, the tracking stride is 2 for frame $k + 2$, and 3 for frame $k + 3$, since the local tracker has to use the last returned result for frame k in performing local tracking for these two frames.

Prior work also examined other optimizations for DNN offloading for AR, *e.g.,* pipelining [9], [10] and frame compression [9], [16], [31]. However, under the network conditions we consider, *e.g.,* 802.11ac, pipelining is less effective as DNN inference latency dominates network delay (§2.8.3), while frame transmission time saved by compression is offset by the compression overhead. To our knowledge, local tracking is the only design option (apart from on-device lite models) that ensures the results for the current frame are available in the current frame interval.

2.3 Motivation

2.3.1 Accuracy Impact of Multitask Offloading

Compared to single-task offlaoding, multitask offloading impacts the task accuracy and hence the app QoE in two ways: (1) it reduces per-task accuracy, and (2) it can reduce accuracy for different tasks by different amount. To quantify these impact, we conducted a measurement study using depth estimation and odometry as the two example tasks. Offloading each task follows the offloading + local tracking paradigm discussed above. The server runs DNN models AdaBins [46] and DAVO [47] for the two tasks, respectively, where a single inference of the two models takes 49 ms and 54 ms on an NVIDIA RTX 2080 Ti GPU. The client runs warping and Kalman filter (see §2.6 for details) as the local trackers for the two tasks. The client and the server are connected with 802.11ac. We report the absolute relative error (AbsRel) for depth estimation and the KITTI odometry metric (t_{err}) for odometry, averaging over all videos in the dataset. The detailed setup, metrics, and dataset can be found in §2.8.1. In single-task offloading, each task is offloaded back to back. In offloading both tasks, we used the simple round-robin and several other static schemes.

Table 2.1. Task accuracies under different offloading schedules.

	Single task	RR	WRR 1:2
Depth estimation (AbsRel $\downarrow$)	0.166	0.192	0.213
Odometry (t_{err} $\downarrow$)	0.038	0.094	0.066
Overall accuracy w/ (0.5, 0.5) weights	N/A	0.143	0.140
Overall accuracy w/ (0.7, 0.3) weights	N/A	0.163	0.169

Impact on per-task accuracy. Table 2.1 shows the average accuracies across all videos in our dataset. Compared to single-task offloading, *i.e.,* only running and evaluating one of the tasks, the accuracies of the two tasks when offloaded under round-robin drop by 15.7% and 166.7%, respectively. Intuitively, the reason for the accuracy drop in concurrent offloading is the reduced offloading frequency, and hence increased local tracking stride. Increased local tracking stride affects the average local tracking accuracy in two ways: (1) frame f_l which corresponds to the latest offloading result becomes more stale and hence less similar compared to the current frame f_c, which leads to less accurate local tracking; (2) more frames, *e.g.,* all frames between f_c and $f_{c+stride-1}$, will be using the stale result (from the server) for f_l in local tracking. The average depth estimation tracking stride increases from 6.68 when offloaded alone to 8.75 when offloaded along with the second task in a round-robin manner.

Impact on relative accuracy drop. We repeated the two-task offloading experiment by changing round-robin to repeatedly offloading depth estimation once followed by offloading odometry twice, denoted as the “WRR 1:2” scheme. Table 2.1 shows compared to round-robin, such an offload schedule improves the accuracy of odometry by 29.8% at the cost of reducing the accuracy of depth estimation by 10.9%. Assume a hypothetical accuracy merge function that calculate the overall accuracy as the weighted accuracy of the two tasks. Table 2.1 shows that under (0.5, 0.5) weights, WRR 1:2 achieves 2.4% higher overall accuracy than RR, while under (0.7, 0.3), RR achieves 3.7% higher accuracy than WRR 1:2.

2.3.2 Prior Work on DNN Offloading

The large amount of prior work on DNN offloading have primarily focused on single-task offloading, and the few exceptions on multi-task offloading did not consider optimizing the current frame accuracy (CFA) of the offloaded tasks.

DNN offloading for a single task. Researchers have designed many DNN offloading systems for single AR tasks such as object detection [9], [16], [31], human pose estimation [9], and depth estimation [10]. Glimpse [16] sends key frames to the server side to perform object detection, and employs local tracking on the client side to mask the offloading latency. Liu *et al.* [9] additionally employ optimizations like pipelined streaming and inference as well as region-of-interest encoding. In addition to visual features, MARVEL [31] utilizes IMU data to track the objects. Meng *et al.* [10] examines the performance of offloading the depth estimation task under different setups, *e.g.,* with or without local tracking and under different bandwidths.

There have also been much work on DNN offloading with a single DNN task with relaxed latency constraints, in particular, for video analytics. These work propose several techniques to optimize the offloading latency. (1) Frame compression algorithms specifically designed for DNN offloading were proposed to improve the compression ratio and task accuracy over standard image/video compression algorithms [19], [48], [49]. (2) Offloading regions of interest [14] or key frames [30] also reduces frame transmission latency. (3) Partial offloading splits the DNN model between the mobile device and the server, at an intermediate DNN layer that is significantly smaller than the input [17], [50]–[54]. Researchers have also studied special cases of the offloading problem, *e.g.,* for high resolution frames [55] and when multiple devices offload similar data [56], [57].

Table 2.2. Comparison between AccuMO and other works on multitask offloading.

	Application	Objective
MCDNN [58]	Video analytics	Max. DFA
LinkShare [59]	Low frame rate apps	Min. deadline violation
AccuMO	AR / latency-critical apps	Max. CFA

Offloading multiple DNN tasks. While most works on optimizing DNN inference for mobile devices have focused on one task at a time, few works studied offloading multiple DNN tasks. Table 2.2 compares existing multitask offloading works with AccuMO. A major difference is that both MCDNN [58] and LinkShare [59] target applications with relaxed latency requirements such as video analytics, while AccuMO focuses on latency-critical applications like AR. LinkShare does not optimize overall accuracy. MCDNN tries to maximize the overall accuracy, but it only focuses on delayed frame accuracy (DFA). In contrast, AccuMO maximizes overall current-frame accuracy (CFA) for AR apps.

Local tracking. Local trackers have been developed for a variety of computer vision tasks — these techniques are orthogonal and can benefit both single-task and multitask DNN offloading. Local trackers for object detection are based on feature point extraction and matching [16], [39], correlation filters [26], [38], [42], motion vectors [9], optical flow [31], and/or inertial data [31]. Human object detection and super-resolution local trackers both utilize motion vectors [9], [40], [41], [43], while the local tracker for depth estimation relies on warping [10].

Optimizing multiple tasks on-device. For on-device DNN inference, Potluck [60] enables computation reuse across applications on the same mobile device via caching, Heimdall [61] coordinates DNN execution and rendering by partitioning and scheduling DNNs in blocks, and RT-mDL [62] schedules a series of compressed models across mobile CPU and GPU based on their accuracy and latency requirements. These systems are all designed for non-real-time applications, and do not meet the latency requirements of AR.

2.4 Key Insights

Intuitively, as the task accuracy is correlated with its offloading frequency, the key to multitask offloading scheduling is to balance offloading of multiple tasks under the constraint of edge server resource, *e.g.,* the GPU. If the accuracy-offloading frequency correlation is static, the optimal offloading schedule can be derived offline. However, we observe that frame content can affect the offloading accuracy. In this section, we discuss two key observations that motivate our adaptive online multitask offloading design.

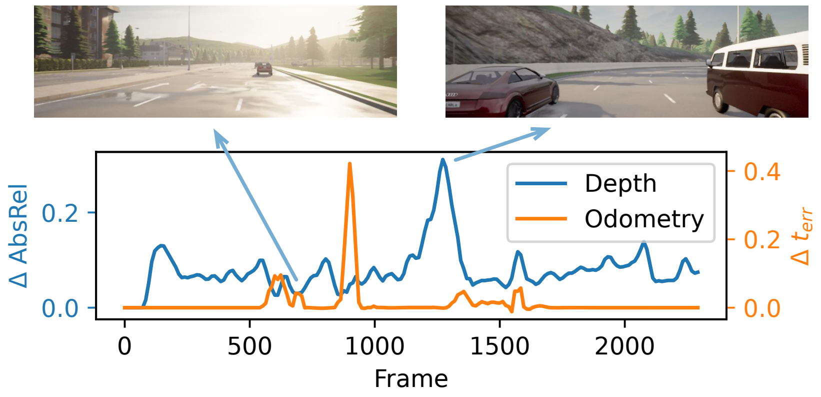


Figure 2.2. Accuracy drop timeline for depth estimation and odometry on a sample video when increasing the stride from 6 to 12.

(1) Offloading frequency-accuracy correlation is frame-dependent. Figure 2.2 shows the local tracking accuracy drop timeline of the two tasks on a sample video when increasing the stride from 6 to 12. From the timeline, we see that local tracking accuracy drops for both tasks change over time. After comparing the accuracy drops with the input frames side by side, we observe that the accuracy drop rate (accuracy drop per frame delay) is affected by the frame content. For example, Figure 2.2 annotates two data points with different depth estimation accuracy drops with their corresponding frames. The one corresponding to high depth estimation accuracy drop has larger regions of cars, because warping — the local tracker for depth estimation — cannot accurately handle regions with moving objects (see §2.6.1). Furthermore, Figure 2.2 shows that there is minimal correlation between the local tracking accuracy drops of the two tasks. In fact, the accuracy drop rate of odometry is instead affected by the angular velocity of the camera (see §2.6.2). We thus make our first observation: *(K1) Local tracking accuracy drop rates are content-dependent, and different tasks may be affected by different content features.*

Design challenges. This observation suggests that instead of offloading the tasks following some static schedule, *e.g.,* round-robin, adaptively increasing (decreasing) a task's offloading frequency when its local tracking accuracy drop rate goes up (down) can potentially result in improved overall accuracy for all tasks. Designing a framework to exploit this observation faces several challenges: (i) how to estimate the local tracking accuracy drop rates during runtime? (ii) how to make the optimal offloading decisions?

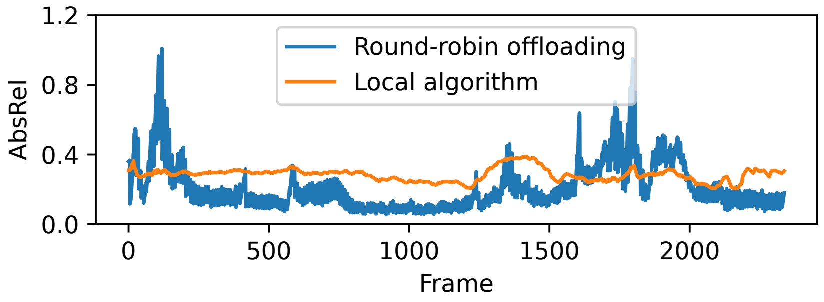


Figure 2.3. Depth estimation accuracy timeline of round-robin offloading vs. the local algorithm FastDepth on a sample video. Round-robin offloading has the same setup as Table 2.1.

(2) Is local tracking the best thing to do on-device? We experimentally compare offloading + local tracking with running *local algorithms*, which are either lite DNN models or conventional algorithms that can finish within one frame interval on the mobile device. Figure 2.3 compares the depth estimation accuracy obtained from round-robin offloading to that of a lite model FastDepth [63] on a sample video. We see that while offloading outperforms the lite model on average (0.212 vs. 0.284), its accuracy is much worse for some frames, *e.g.,* frames 1600 to 2000. We make our second key observation: *(K2) Even though the accuracies of the local algorithms are significantly worse than the offloading solutions on average across the frames of a video, the local algorithms can achieve better accuracies for some individual frames.* In particular, this happens in two situations: (1) when the content changes very fast, and (2) when the local tracking stride is large enough, *i.e.,* due to other offloading tasks occupying the server GPU resource. Further, we observe little correlation between offloading and lite model accuracies. This is because while the local tracker is affected by content changes, the lite model, which is a DNN model taking a single frame as input, is more likely affected by static features within a frame.

Design challenges. Exploiting K2 in multitask offloading faces several challenges: (i) how to estimate the local algorithm accuracies during runtime? (ii) how to incorporate the accuracy estimates in making offloading decisions?

2.5 AccuMO Design

Motivated by the above key insights, we design a content-aware adaptive multitask offloading framework called AccuMO that jointly optimizes the accuracies of multiple tasks by dynamically controlling each task's offloading frequency and switching between offloading and local algorithms.

2.5.1 Design Goals

We design the multitask offloading framework to achieve the following goals:

Support for different application needs. The system should accommodate different high-level application needs, in particular different needs for balancing task accuracies.

Optimized overall task accuracy. The system should optimize the overall accuracy across the tasks.

Extensible for different tasks. The system can easily support varying numbers of tasks. Furthermore, it should accommodate different task designs, *e.g.,* with or without a local algorithm.

Real-time. The system should produce results in real-time, *e.g.,* 60 FPS, to support latency-critical applications like AR.

2.5.2 Architecture Overview

Design rational. As shown in Figure 4.4, AccuMO employs a modular design to support different numbers of tasks. There is one task component for each task. To meet the real-time requirement, we employ the offloading + tracking paradigm for each task, and thus

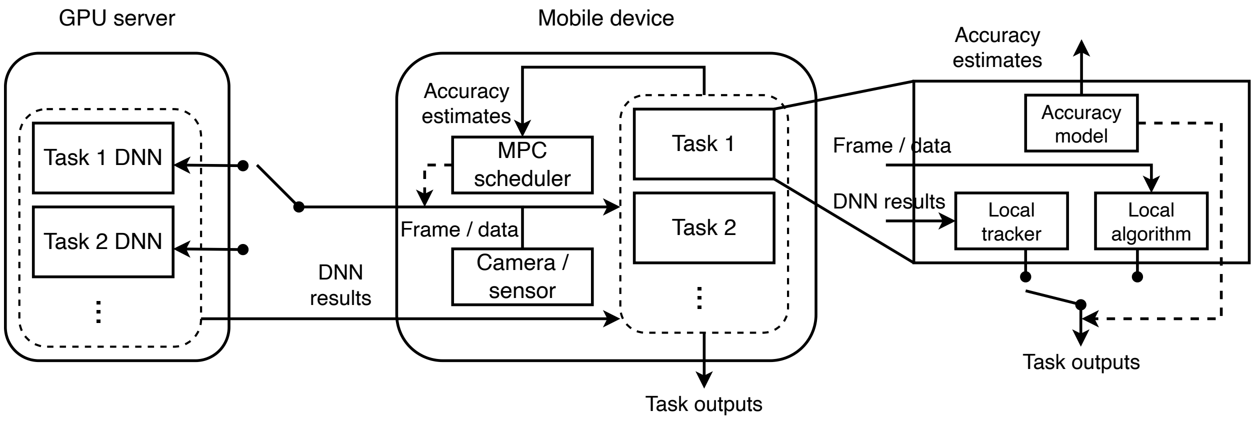


Figure 2.4. AccuMO architecture. Solid arrows represent data flows, while dashed arrows represent control flows. Inputs to accuracy models are task-specific and not shown.

each task component requires a high accuracy DNN model running on the server GPU and a local tracker that runs in real-time. Furthermore, to exploit our second key insight (K2), each task component may also contain a local algorithm. To dynamically switch between the two, an accuracy model is required to estimate their accuracies based on the frame content.

Exploiting our first key insight (K1), we employ a global scheduler to control when and for which task to offload each frame captured by the camera and/or other sensors based on each task's accuracy estimates. We design our global scheduler using model predictive control (MPC) [27], a control theory optimization algorithm, rather than machine learning algorithms, to meet our flexible accuracy and extensibility design goals. Machine learning algorithms would require hardcoded task number, task design, and optimization goals which then have to be trained offline, while control theory-based algorithms are more flexible in dynamically adapting to different tasks and application needs, *e.g.,* how to merge the task accuracies.

Our MPC scheduler only decides on the offloading schedule, while leaving the decision of switching between offloading and local algorithms to the individual tasks. While an algorithm that makes both decisions jointly might produce better schedules, we decide to decouple the decisions to accommodate diverse task component designs. For example, a task might not have a local algorithm, or a single design may serve as both the local tracker and the local algorithm and the "local tracker vs. local algorithm" decision is made internally (see §2.6.2). Our MPC scheduler accommodates both of these cases by simply not considering the local algorithm accuracy drop for that task.

Control loops. We develop a modular design that runs two concurrent control loops that both adapt according to frame content dynamics: a low-level control loop is per task and adapts between local tracking and using the local algorithm for each frame; and a global control loop runs MPC to dynamically balance offloading of the tasks. We observe that accuracy models may be time consuming to run, or they can only estimate task accuracies on certain key frames (see §2.6.1). To meet the real time and extensibility goals, we run accuracy models concurrently with the other two control loops. Effectively, the two control loops make decisions based on the most recent accuracy estimates, exploiting the temporal locality. In summary, the framework consists of two control loops, in addition to the accuracy models, running concurrently and continuously:

- **MPC scheduler:** When a new frame becomes available, and there is currently no unfinished offloading, the MPC scheduler determines the next task to offload for that frame based on the latest accuracy estimates by the task-specific accuracy models.
- **Local selector:** For each frame, either the local tracker or the local algorithm is executed for each task, depending on which one has higher estimated accuracy.
- **Accuracy model:** The local tracker and local algorithm accuracy models for each task are executed repeatedly on the most recent frame, *i.e.,* best effort. The most recent accuracy estimates are used by the two control loops.

Adding new tasks. Our modular design simplifies the addition of new tasks. To offload a task under any framework under the offloading + local tracking paradigm, the developer already needs to choose and prepare a local tracker to run on the client, and a DNN model to run on the server. To add such a task in AccuMO, the developer (1) can reuse the chosen local tracker and server DNN model, (2) optionally uses an off-the-shelf local algorithm for the task (*e.g.,* a lightweight DNN model), and (3) only needs to develop an accuracy model for the new task. We provide a set of easy-to-follow guidelines for developing accuracy models in §2.5.4, and we show how to apply them to derive the accuracy models with a case study in §2.6.1 and §2.6.2.

2.5.3 Local Tracker and Local Algorithm

As described in §2.3, local trackers adjust the stale results from the server DNN models for the current frame. They need to run fast in order to be executed for every frame. For many tasks, off-the-shelf algorithms, possibly designed for other purposes originally, could be used readily [10], [26], [44].

In contrast to the local tracker, a local algorithm directly produces task results from camera or sensor data, without relying on server DNN results. It needs to be fast to be executed for every frame. The local algorithm could be a lite DNN model designed for mobile devices, but it could also be a conventional algorithm without any learning components.

2.5.4 Accuracy Model

The accuracy models for each task estimate the accuracies for both offloading (*i.e.,* the local tracker) and the local algorithm. First, to model the accuracy for offloading, since local trackers introduce accuracy drops in adjusting the stale server DNN results, we only need to estimate the accuracy drop (w.r.t. that of running the server DNN model) or accuracy drop rate (per frame delay). We propose a set of principles that can be easily followed in developing accuracy models for different tasks. (1) We start by identifying limitations of the local tracker, *i.e.,* what kind of inputs lead to bad tracking results. (2) Next, we identify and extract features that quantify good inputs vs. bad inputs. (3) Finally, we derive the correlation between the extracted features and accuracy drops or accuracy drop rates.

Second, for modeling the accuracy of local algorithms, since they typically have much worse accuracy compared to the server DNN models, an effective method to approximate their accuracy is by treating the server DNN results as the ground truth [9], [64]. To make the accuracy modeling comparable to local trackers' estimated accuracy drops (as opposed to accuracy), we subtract the local algorithm accuracy estimate by the server DNN model's average accuracy (calculated offline), which transforms it into the accuracy drop relative to the DNN model. §2.6.1 shows such an accuracy estimation method on the depth estimation lite model works well. However, other task-specific methods may also be used for local algorithm accuracy estimation.

2.5.5 Model Predictive Control Scheduler

Ideally, given perfect knowledge of offloading and local algorithm accuracies over an entire video, the optimal offloading plan can be calculated offline.[1] While perfect accuracy information is not available in practice, it is possible to estimate the current and future accuracies during a short horizon of N frames $[k, k+N]$ using the accuracy models described before. With the estimated accuracies, we can calculate the optimal offloading plan in this horizon, apply the first step of the plan (*i.e.,* for the current frame), and move the horizon forward to $[k+1, k+N+1]$. This scheme is known as model predictive control (MPC) [27], [65].

Algorithm 1: Offloading scheduling using MPC

```
input: the list of tasks $T$ with offloading histories
       latest accuracy estimates $A$ for all tasks
       horizon length $N$
       overall accuracy function $accuracy\_merge()$
1 $d^* = \infty, t^* = t_{\text{default}}$;
2 for $p$ in ValidOffloadPlans$(T, N)$ do
3     $d = $ SimulatePlan$(p, T, A, accuracy\_merge())$;
4     if $d < d^*$ then
5         $d^* = d, t^* = p[0]$;
6 offload the current frame for task $t^*$;
```

The MPC algorithm. Algorithm 1 shows the MPC offloading scheduling algorithm for deciding the next task to offload the frame/sensor data for. The algorithm is executed to decide the offloading task whenever a new frame is to be offloaded, *i.e.,* when the last offloaded result has been received by the mobile device. It essentially populates and simulates all valid offloading plans within a horizon of N frame intervals, calculates the overall accuracy

[1]↑We assume the network transmission delay is relatively stable, which holds under the network conditions and frame sizes we consider (§2.8.1).

drop of each plan using the accuracy estimates from the accuracy models, and picks the first offloaded task in the plan that has the lowest overall accuracy drop in the horizon for offloading.

Since the maximum offloading frequency is constrained by the task offloading latencies, only the plans that offload the next frame after the last offloading results have come back are valid. This drastically reduces the number of plans to be simulated and allows the MPC algorithm to run in real-time. Further, we prune heavily unbalanced plans where one task is offloaded much more often than the other, *e.g.,* one task is offloaded four times while the other is offloaded only once, which leads to overly unbalanced task accuracies. Each valid plan is simulated by starting with the current task offloading status and rolling out the steps in the plan. For each frame, we choose the lower between the local tracker's accuracy drop and the local algorithm's accuracy drop. Finally, we compute the accumulated overall accuracy drop.

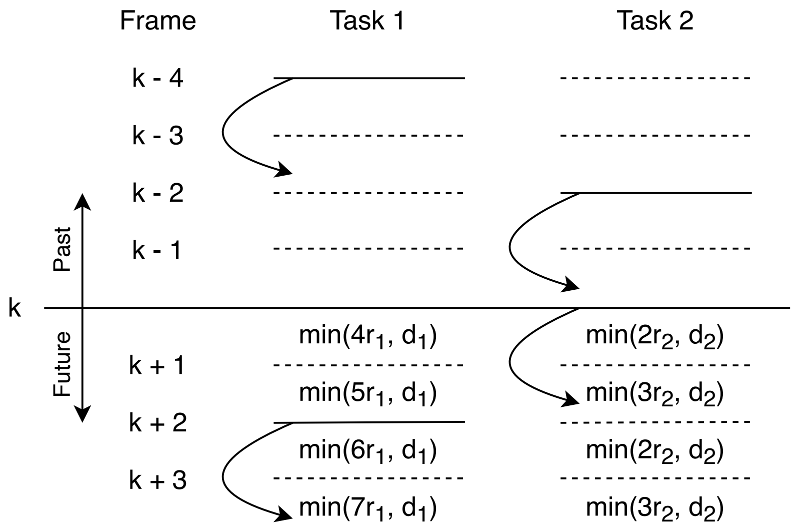


Figure 2.5. Simulating an example MPC plan [2, 0, 1, 0]. A curved arrow points from a frame offloaded for a task, to the frame interval when the offloading result comes back.

An example. Consider an application with two offloading tasks, and the offloading latencies are two frame times for both tasks. With a horizon $N = 4$, the valid plans are [1, 0, 1, 0], [1, 0, 2, 0], [2, 0, 1, 0], and [2, 0, 2, 0], where 1 and 2 represents offloading task 1 and task

2 respectively, and 0 means no offloading. Assuming the local tracker accuracy drop rates per frame are r_1 and r_2 for the two tasks, the local algorithm accuracy drops are d_1 and d_2, and frame $k-4$ was offloaded for task 1 while frame $k-2$ was offloaded for task 2. Figure 2.5 shows the simulation of the plan [2, 0, 1, 0]. The local tracker accuracy drop is calculated by multiplying the tracking stride with the local tracker accuracy drop rate, *e.g.,* $4r_1$ in $\min(4r_1, d_1)$. Finally, the plan's overall accuracy drop is obtained by merging the accuracy drops of all tasks across all frames in the plan according to *accuracy_merge()*.

2.6 Case Study: Offloading AR Tasks

In this section, we present a case study of how the modular design of our AccuMO framework can easily support multiple tasks, using two representative tasks from a complete AR app — depth estimation and visual odometry. Since the MPC-based offloading scheduler is generic, applying AccuMO in our case study boils down to designing the local tracker, the local algorithm, and their accuracy models for each task.

2.6.1 Depth Estimation

Background. Depth estimation infers the depth map for a given image, containing the distance between the camera and surrounding environment represented by each pixel. The depth estimation results can be used for many tasks, *e.g.,* rendering occlusion between virtual and physical objects in AR, and perception in self driving. In this chapter, we focus on monocular depth estimation, since most smartphones are equipped with monocular cameras. Recently, several monocular depth estimation DNNs have been proposed (*e.g.,* [46], [66], [67]). While accurate, such DNN models are compute-intensive and require offloading.

Depth local tracker. We use warping [10], a lightweight geometry-based algorithm, as the local tracker for the depth estimation task. It takes as input the depth map of the last frame D_1, and the relative pose change between the last frame and the current frame p_{12}, and outputs the depth map of the current frame D'_2 as an approximation for the unknown D_2. Specifically, given the intrinsics parameters of the camera, warping first converts the depth map into a point cloud in the last frame's camera coordinate system. Then, using p_{12}, it transforms the point cloud into the current frame's camera coordinate system. Finally, it projects the point cloud onto the camera plane and performs nearest interpolation to generate D_2', the estimated depth map of the current frame.

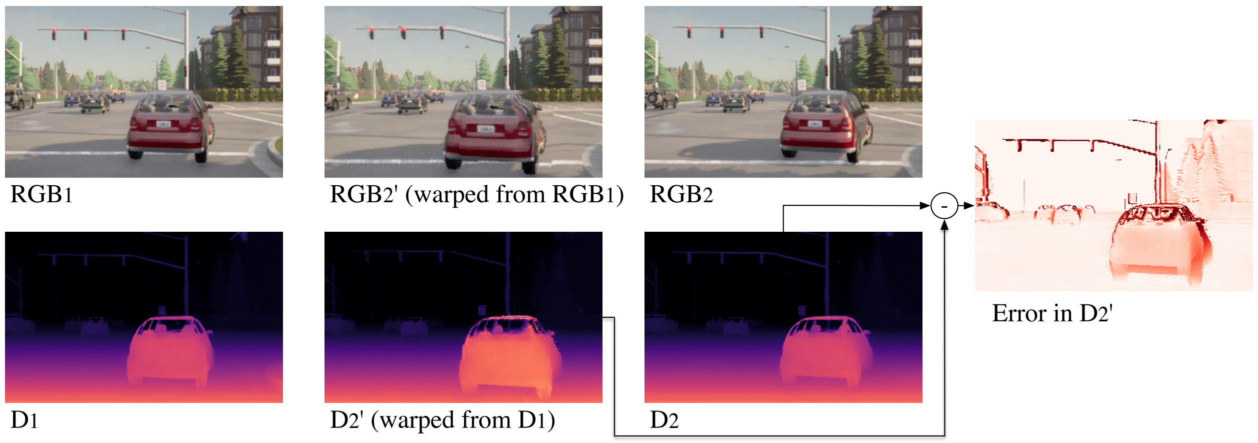


Figure 2.6. Moving objects cause errors for depth local tracker.

Accuracy model for depth local tracker. The depth accuracy model estimates the error increase caused by the local tracker, *i.e.*, warping, relative to if the server DNN model could be used. Since warping assumes the objects in the scene are static, dynamic objects will make warping inaccurate. This is shown in Figure 2.6, where a second vehicle is moving in front of the ego vehicle where the camera sits. Warping assumes that the second vehicle is stationary relative to the ego vehicle and hence the camera, and thus the vehicle in D_2' is incorrectly predicted as being closer to the ego vehicle. This results in high errors around the second vehicle, as shown in the error map. We make a key observation that the magnitude of the error is determined by the relative depth between the vehicle's surface and the background, and hence the error of D_2' can be estimated by (1) identifying regions in the frame that are affected by moving objects; and (2) accounting for the relative depth difference between the moved object and the background.

Based on the observation, we propose a novel accuracy model for estimating the warping error: (1) Warp RGB_1 to RGB_2'. Since warping assumes objects are static, RGB_2' differs from the real RGB_2 on moving objects. (2) Capture moving objects by calculating the optical flow $\mathbb{F}$ from RGB_2' to RGB_2, and (3) compute D_2'' by translating the pixels of D_2' based on $\mathbb{F}$, where the resulting D_2'' accounts for the motion of moving objects. (4) Finally, calculate the absolute relative error (AbsRel) between D'_2 and D''_2 as an estimate for the error caused by warping. Note that we are using AbsRel between D'_2 and D''_2 as a proxy for the AbsRel between D'_2 and the ground truth. Assuming the warping error increases at a constant rate, we divide the estimated error by the warping stride between the two input frames, to get the error increase rate r (the average error increase per frame).

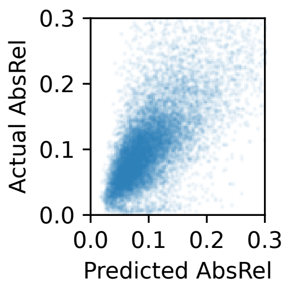


Figure 2.7. Predicted and actual depth estimation tracker error.

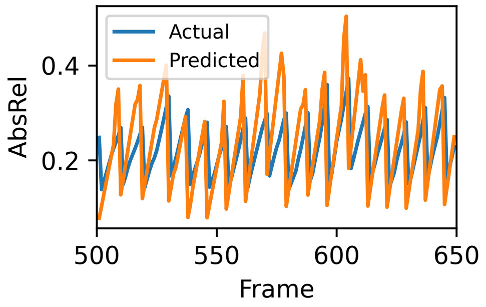


Figure 2.8. Example timeline of the accuracy model. Sharp drops are due to receiving offloaded results.

Figure 2.7 shows the predicted and actual depth estimation errors exhibit a strong correlation using the proposed depth accuracy model. Figure 2.8 plots an example timeline which shows the predicted error follows the actual error closely.

Local algorithm and its accuracy model. Since the above depth estimation DNN models [46], [66], [67] are compute-intensive, several lightweight depth estimation DNN models have been proposed to run on mobile phones and embedded devices in real-time [63], [68], with compromised accuracy.

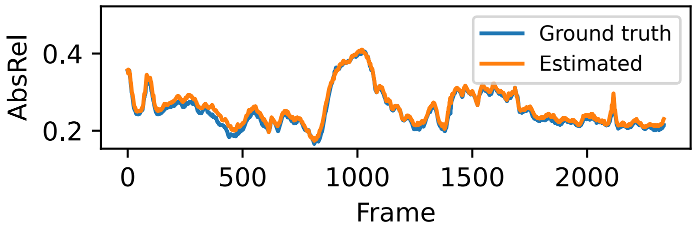


Figure 2.9. Example timeline of estimated error vs. ground-truth error of the depth estimation local algorithm.

To support adaption between local tracking and lite model, we estimate the error of the lite model following the method described in §2.5.4. The client estimates the error of the lite model's output against the server returned depth map for the same frame. Since the server DNN model is much more accurate, the estimated error against server DNN output is close to that against the ground truth, as shown in Figure 2.9.

2.6.2 Odometry

Background. Odometry is the use of various sensor data to estimate the ego-pose (location and orientation) of the device relative to a starting position. The input sensor data can be monocular or stereo camera frames, depth maps (*e.g.,* from Lidar), IMU, GPS, or a combination of them. We focus on commodity smartphones in indoor or urban areas, where only monocular camera frames and IMU data are available. For camera frames, a pair of consecutive frames f_1 and f_2 are taken as inputs, and the algorithm estimates the relative pose change between f_1 and f_2. The current pose is estimated by accumulating the sequence of pose changes from start. In offloading, f_1 and f_2 are offloaded frames, which could be non-consecutive. The poses of the frames in-between are given by the local tracker, as described below.

Kalman Filter as Local Tracker and Local Algorithm. We use a Kalman filter [69] together with the IMU data as both local tracker and local algorithm for odometry. Since IMU only reports accelerations and angular velocities, the Kalman filter first integrates them to obtain translations and rotations. Secondly, the Kalman filter maintains internal states including position, velocity, orientation, and their estimated error covariance, which helps to reduce IMU sensor noise. For example, if the velocity covariance is small yet the acceleration reported by IMU is huge, it is likely that the acceleration data contain noise. Lastly, the server DNN pose estimations can be similarly fused with the Kalman filter's internal states. Hence the Kalman filter with IMU data can estimate ego-pose with or without server DNN results, and we use it as both a local tracker (when with server DNN results) and a local algorithm (when without) in our framework. Furthermore, since it calculates the relative importance between server DNN results and IMU data based on the error covariance, it is also used as an accuracy model for choosing between offloading and local algorithms.

Accuracy Model. Since the Kalman filter does not explicitly estimate the local algorithm accuracy drop, we will design an offloading accuracy model for use in the MPC scheduler. We first introduce the odometry accuracy metric.

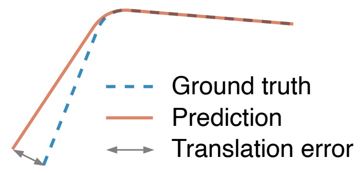


Figure 2.10. Translation error between trajectory segments.

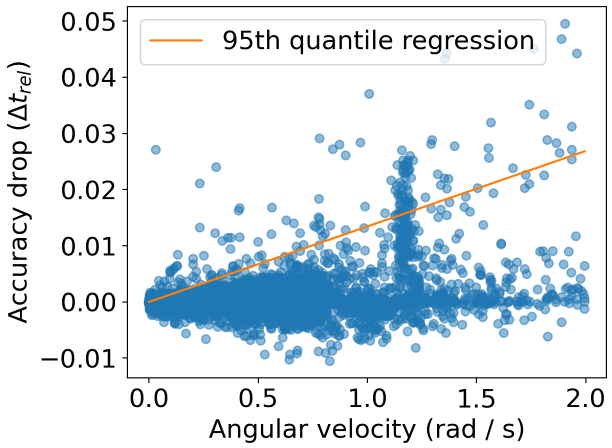


Figure 2.11. Scatter plot of accuracy drop vs. angular velocity.

Accuracy metric. The commonly used KITTI odometry metric [70], [71], denoted as t_{err}, is calculated in 3 steps: (1) extract all pairs of segments of (100, 200, ..., 800) meters long from both ground truth and prediction trajectories, (2) compute the translation error between each pair (see Figure 2.10) and divide it by the segment length, and (3) average the error over all segment pairs.

The above odometry metric t_{err} is calculated offline with segment as the basic unit, which cannot be used by the MPC scheduler in our framework which plans online at the frame level. To this end, we adapt t_{err} for the MPC scheduler and calculate the accuracy drop caused by a different offloading schedule on a sequence of frames by calculating t_{err} only over the segments that contain the frames, while keeping the offloading schedule for other frames unchanged. We use this frame-level accuracy as a proxy for the offline t_{err} in the MPC scheduler instead.

Camera rotation amplifies translation error. By comparing the trajectories produced by different offloading schedules, we observe that they mainly deviate when the camera rotates (*i.e.,* with high angular velocity, see Figure 2.16 for an example). The reasons are two-fold. First, camera rotation changes the view more drastically compared to translation, and thus it is more difficult to match the two input frames as the stride increases, where the stride for the odometry task is the difference between the frame IDs of the two consecutive offloaded frames. Second, as shown in Figure 2.10, the translation error of a frame is amplified when it is further away from the camera rotation point, causing higher t_{err}.

The accuracy model. We draw the scatter plot between angular velocity and accuracy drop in Figure 2.11 to quantify the correlation between them. The accuracy drop is calculated when increasing the inter-frame stride from 4 to 8 for a sequence of 32 frames while keeping the rest at stride 4. The angular velocity is calculated on the same 32 frames. The data points are mainly distributed at the bottom region, rather than forming a line, as accuracy drop depends on not only the 32 frames, but also all the frames in the affected segments. Nonetheless, Figure 2.11 indicates that angular velocity can be used to estimate the upper bound of accuracy drop.

Furthermore, we argue that it is imperative to optimize all the frames with high accuracy drops, as these frames cause high errors to all segments that contain them, which spread the impact of suboptimal offloading schedule to longer distances, which was not captured by the proxy accuracy drop calculation (see §2.8.5). For this reason, we estimate the upper bound of the accuracy drops. Since the upper bound can be affected by outliers, we perform the 95th quantile regression [72] which corresponds to the orange line in Figure 2.11.

During runtime, the angular velocity is calculated from recent poses, and the upper bound of offloading accuracy drop is estimated based on the model discussed above, which is derived offline. Similarly, we assume linear accuracy drop with increasing strides in estimating the accuracy drop rate.

2.7 Implementation

We implement our multitask offloading framework AccuMO for the case study on depth estimation and odometry.

Client. We implement the client, including the MPC scheduler, task-specific local trackers, local algorithms, and accuracy models as an Android application in 2K lines of Java and C++ code. The control loops and the MPC scheduler are implemented in Java. For depth estimation, the local tracker (warping) is implemented in C++ and runs on the phone CPU. We use FastDepth [63], a state-of-the-art depth estimation DNN designed for embedded devices, as the local algorithm, and run it on the phone CPU on top of the ncnn [73] DNN inference framework. We use FlowNet2S [74], a lightweight DNN, to estimate the optical flow used by the accuracy model, and it runs on the phone GPU utilizing TensorFlow Lite [75]. For odometry, the Kalman filter implementation adopts the `insfilterErrorState` object in the MATLAB Sensor Fusion and Tracking Toolbox [76], and is converted to C using the MATLAB Coder [77] to run on mobile devices.

Server. We implemented the server in about 300 lines of Python code. We use AdaBins [46] (implemented in PyTorch) as the server DNN model for depth estimation and DAVO [47] (implemented in TensorFlow) for odometry. Both DNN models are loaded upon server startup.

The client and the server communicates over TCP. Camera frames are uploaded to the server in raw YUV420 format (the default Android camera format) with resolution 448×128, while depth maps are sent back in raw 16-bit bitmaps.

2.8 Evaluation

2.8.1 Methodology

Evaluation setup. Our client app runs on a Samsung Galaxy Note20 Ultra 5G phone with a Qualcomm Snapdragon 865+ SoC, which has eight Kyro 585 CPU cores and an Adreno 650 GPU. We use two different servers with GPUs of different tiers, an NVIDIA GeForce RTX 2080 Ti and a more powerful NVIDIA A40. We evaluate AccuMO under 802.11ac for indoor scenarios and 5G mmWave for outdoor scenarios. For 802.11ac, we connect the phone and the server to the same access point (550 Mbps for both uplink and downlink, 3 ms RTT). We emulate the 5G mmWave network using the tc tool on top of the 802.11ac setup, based on recent 5G mmWave measurements (152 Mbps uplink, 1715 Mbps downlink, 14 ms RTT) [12]. However, note that we are only able to emulate up to 550 Mbps for 5G mmWave downlink due to the 802.11ac downlink bandwidth limit. When not specified, we default to the 2080 Ti GPU and 802.11ac network.

Dataset. To evaluate our case study of performing depth estimation and odometry for AR, we need a dataset that concurrently provides IMU data and ground truths for both depth estimation and odometry, and has a camera frame rate of at least 60 FPS. To the best of our knowledge, we are not aware of any existing dataset that satisfies all these requirements. To this end, we resort to the same methodology as in prior works [78], [79] — we use CARLA [80], an autonomous driving simulator, to generate a synthetic dataset with sufficient sensor data and ground truth labels. We generated videos of 40 seconds long with the camera mounted on top of the ego-vehicle capturing frames of size 832×256 at 60 FPS. As the simulator only outputs ground truth IMU data, we added noise to the IMU data based on typical IMU data sheets [81].

To analyze the impact of video content, we further classify videos into "easy" ones vs. "hard" ones. In §2.6, we show that depth estimation offloading accuracy is affected by moving objects, while odometry is affected by camera rotation. To this end, we categorize the videos based on (1) the percentage of pixels occupied by dynamic objects identified by a semantic segmentation DNN [82], and (2) the average camera angular velocity using ground truth pose. We classify each video to have high (H) / low (L) dynamic object percentage / angular velocity, and select 20 videos from each of the 4 combinations, *i.e.*, L/L, L/H, H/L, and H/H. We generated 20 additional videos to train the DNN models (see §2.7).

Metrics. We evaluate depth estimation using the popular absolute relative error (AbsRel) [83]. We evaluate odometry using the KITTI odometry metric (t_{err}), as discussed in §2.6.2.

2.8.2 Baselines

We compare AccuMO with the following baselines:

Local algorithms (LA). In this setup, all task results are produced by the local algorithms running on the phone without offloading.

Round-robin (RR). This setup offloads the tasks in a round-robin manner (Figure 2.1b).

Weighted round-robin (WRR). This setup is similar to RR, except that some tasks are offloaded more frequently than others. We denote a specific configuration of WRR as WRR x:y, where the first task (depth estimation) is offloaded x times and then the second task (odometry) is offloaded y times. We study WRR 3:1, 2:1, 1:2, and 1:3.

All at once (AAO). In this setup, for each offloaded frame, DNN models for all tasks will be executed.

In RR, WRR, and AAO, the offloading results are processed by the local trackers, but local algorithms are not used. We will evaluate the impact of local algorithms in §2.8.5.

We do not provide direct comparisons to MCDNN [58] and LinkShare [59], as they target different applications and have different design objectives, and their mechanisms are not applicable to AR applications, as discussed in §3.7.

2.8.3 Overall Performance

We compare the performance of AccuMO with all baselines under the default environment setup and a number of accuracy merge functions that calculate weighted average of the accuracies for depth estimation and odometry tasks with varying weights 4:1, 2:1, 1:1, 1:2, and 1:4. We set the MPC horizon to be 30 frames, and restrict the maximum task offloading ratio to be 3:1, *i.e.,* the other task will be offloaded next time if one task has been offloaded three times in a row.

Comparison with baselines. We compare the overall accuracy of our framework, which is the weighted average of the task accuracies, with other baselines under different accuracy weights. As shown in Figure 2.12, AccuMO improves over the best baseline (the one with the best average overall accuracy across the four video categories) under each weight ratio 4:1, 2:1, 1:1, 1:2, and 1:4 by 2.3%–11.9% (avg. 7.6%), 6.7%–14.6% (avg. 10.1%), 10.8%–16.7%

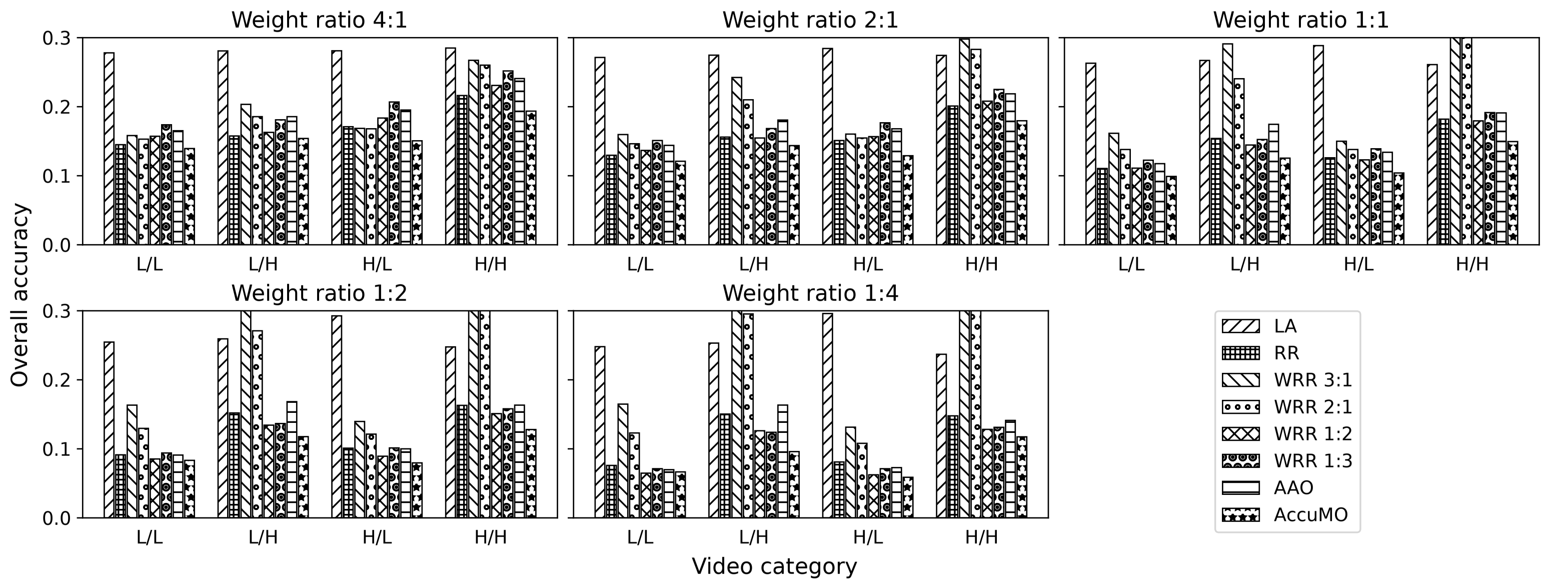


Figure 2.12. The overall accuracies (lower is better) under different video categories and varying accuracy weight ratios.

(avg. 14.3%), 2.3%–15.2% (avg. 11.2%), and -2.8%–23.9% (avg. 11.2%) across different video categories. In particular, AccuMO performs on par with the best baseline, *i.e.,* WRR 1:2, in worst cases (-2.8% improvement on L/L videos for weight ratio 1:4), while it outperforms the best baseline significantly in best cases (23.9% improvement on L/H videos for weight ratio 1:4).

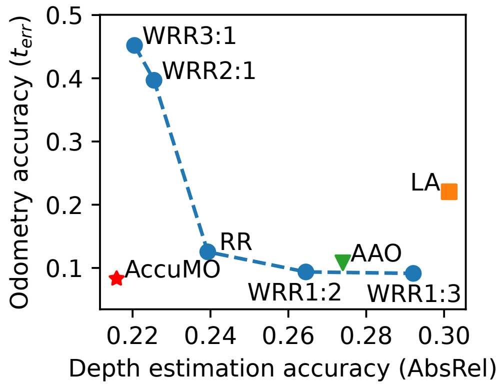


Figure 2.13. The accuracy tradeoffs between the two tasks on H/H videos.

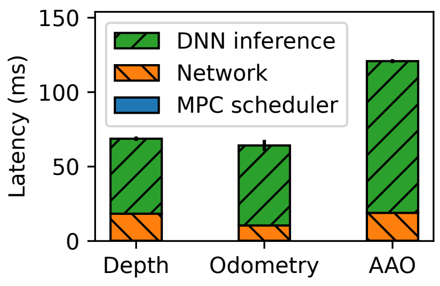


Figure 2.14. The offloading latency breakdown of the two tasks, and AAO.

Individual task accuracies. While the objective of AccuMO is to optimize the overall accuracy, individual task accuracies help us understand why a scheduler has good or bad overall accuracy. Table 2.3 shows the accuracies of individual tasks under weight ratio 1:1, along with the overall accuracy. We make the following observations. (1) The accuracies of LA for both tasks are among the worst across all algorithms, indicating the need for offloading. (2) For the RR and WRR schedulers, as shown in Figure 2.13, offloading one task more frequently improves the accuracy of that task, but hurts the accuracy of the other task. (3) In contrast, AccuMO consistently improves over RR, obtaining 4.3%–13.4% accuracy improvements for depth estimation and 18.9%–33.8% accuracy improvements for odometry, depending on the video categories. (4) Finally, AAO represents a different kind of tradeoff compared to the RR and WRR schedulers. The depth estimation accuracy drops due to increased tracking stride, while the odometry accuracy improves due to decreased stride between the two odometry DNN input frames. However, our framework still outperforms

Table 2.3. Average depth (estimation) error (AbsRel), odom(etry) error (t_{err}) under weight ratio 1:1, along with overall accuracies (lower is better). The best accuracies achieved under each video category are highlighted in **bold**.

	L/L			L/H			H/L			H/H		
	Depth	Odom	Overall	Depth	Odom	Overall	Depth	Odom	Overall	Depth	Odom	Overall
LA	0.288	0.238	0.263	0.290	0.244	0.267	0.276	0.301	0.289	0.301	0.221	0.261
RR	0.168	0.053	0.111	0.160	0.148	0.153	0.201	0.051	0.126	0.239	0.125	0.182
WRR 3:1	0.156	0.167	0.161	**0.145**	0.437	0.290	0.181	0.119	0.150	0.221	0.452	0.336
WRR 2:1	0.163	0.113	0.138	0.149	0.332	0.241	0.188	0.088	0.138	0.226	0.397	0.311
WRR 1:2	0.188	**0.034**	0.111	0.175	0.114	0.145	0.224	**0.022**	0.123	0.265	0.094	0.179
WRR 1:3	0.208	0.037	0.123	0.200	0.105	0.152	0.252	0.026	0.139	0.292	0.091	0.192
AAO	0.197	0.038	0.117	0.193	0.156	0.174	0.236	0.032	0.134	0.274	0.108	0.191
AccuMO	**0.155**	0.043	**0.099**	0.153	**0.098**	**0.126**	**0.174**	0.034	**0.104**	**0.216**	**0.083**	**0.149**

AAO, as our dynamic offloading schedule can provide even smaller strides for frames with high accuracy drop rates.

Impact of video content. From Table 2.3, we observe that the content or the difficulty of the videos have huge impacts on both task accuracies and task accuracy drop rates. Videos with higher percentage of dynamic objects lead to overall worse depth estimation accuracies, and the accuracy differences among the RR and WRR schedulers are also higher. The same relationship applies to videos' average angular velocity and the odometry accuracy. For both tasks, our framework achieves higher accuracy improvements over RR on more difficult videos due to the higher accuracy drop rates. For example, for depth estimation, our framework improves over RR by 7.7% on L/L videos but by 13.4% on H/L videos. In general, a task is offloaded more frequently when its accuracy drop rate is high and the other task's is low. However, we note that our framework is still able to improve the accuracies of both tasks by 9.6% and 33.6% respectively on the H/H videos. As we will see in §2.8.4, the reason is two-fold. First, the offloading accuracy drop rates for both tasks still change over time, and they change independently. Secondly, the local algorithms can be used instead when both tasks have high offloading accuracy drop rates, which mitigates the contention on server resources.

Latency breakdown. Figure 2.14 breaks down the end-to-end offloading latency under AccuMO for both tasks and when performing AAO offloading. Offloading a frame for depth estimation and odometry takes 70.98 ms and 61.05 ms respectively, which translates to 5 and 4 frame times. For both tasks, server DNN inference (50.39 ms and 53.73 ms) takes up a major portion of the total latency. For network transmission, the majority of the time for odometry is spent uploading the frame to the server, while for depth estimation, more time is needed to send the depth map back. We also note that our MPC scheduler is fast (0.2 ms) and has minimal impact on offloading latency. Finally, the AAO offloading latency is much longer than offloading a single task, as both server DNN models need to be executed.

As for other components in the case study, the depth estimation local tracker (warping) takes 15.04 ms, the depth estimation local algorithm (FastDepth) takes 13.42 ms, and the odometry local tracker/local algorithm (Kalman filter) takes 0.03 ms, *i.e.,* all finish within one frame time, making it possible to support real-time applications like AR. For the accuracy models, while the odometry accuracy model (extracting angular velocities) and the depth estimation local algorithm accuracy model (comparing the local algorithm depth map with the server DNN depth map) take minimal time, the depth estimation offloading accuracy model (estimating the optical flow using FlowNet) takes 66.15 ms, and thus it has to run asynchronously, as discussed in §2.5.2.

2.8.4 Scheduler Behavior Analysis

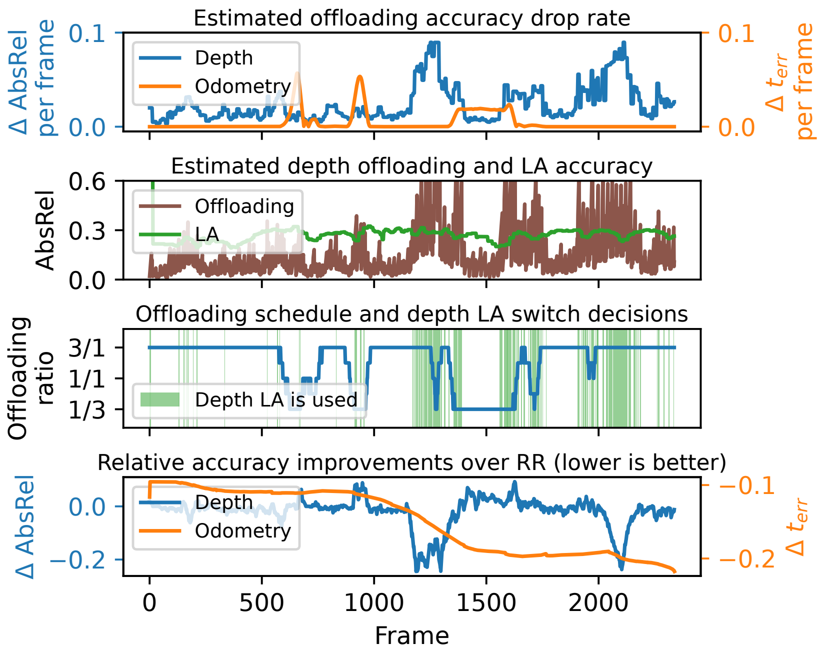


Figure 2.15. The timeline for a sample H/H video.

To understand how the estimated accuracy drops affect AccuMO's scheduling decisions, and how the scheduling decisions in turn affect the task accuracies, in Figure 2.15 we plot all the relevant information in the same figure for a sample H/H video.

Similar to what is shown in Figure 2.2, the estimated accuracy drop rates of the two tasks are largely uncorrelated (1st figure). The MPC scheduler makes offloading decisions based on the estimated accuracy drop rates. The 3rd figure plots the depth estimation vs. odometry offloading frequency ratio within a sliding window of 8 offloaded frames. As we restrict the maximum offloading ratio to 3:1, the ratio ranges from 1/3 (when odometry is offloaded frequently) to 3/1 (when depth estimation is offloaded frequently). We see that a task is offloaded more frequently when its accuracy drop rate is relatively high and the other task's is relatively low. Furthermore, we notice that the offloading ratio stays at 3/1 frequently, which means that odometry is offloaded less frequently than depth estimation. However, the odometry accuracy is still better than that of RR, which we explain in §2.8.5. When the depth estimation local algorithm gives better accuracy over offloading as shown in the 2nd figure, *e.g.,* for frames 1200–1400, the framework switches to local algorithm instead for depth estimation (3rd figure). Meanwhile, the offloading opportunities are saved for odometry, as indicated by the close to 1/3 scheduling ratio in the 3rd figure. The 4th figure shows that the depth estimation accuracy, which is the moving average of 16 frames, improves over RR when the offloading frequency increases, *e.g.,* for frames 200–600, or when the local algorithm is used, *e.g.,* for frames 1200–1400, while the accuracy is worse than RR when the offloading opportunity is saved for odometry, *e.g.,* for frames 900–1000. On average, the depth estimation accuracy is better than that of RR.

Impact of accuracy model. The optimality of AccuMO's scheduling decisions could be affected by the accuracy of the accuracy models. To estimate the impact, we test AccuMO with accuracy model outputs replaced by actual accuracy drops. The overall accuracy on H/H videos under weight ratio 1:1 is 0.148, on par with that of AccuMO backed by accuracy models (0.149). This is because AccuMO only requires accuracy models to have moderate accuracy levels that are sufficient for the MPC scheduler to determine the right offloading ratios (3rd figure of Figure 2.15), and the experiment shows that our accuracy models are adequate to support AccuMO's scheduling decisions.

2.8.5 Task-Specific Analysis

Depth estimation. Figure 2.15 has shown that the depth estimation accuracy improvement over RR is contributed by two factors: (1) the depth estimation task is offloaded more frequently when its estimated accuracy drop rate is high, and (2) the local algorithm is instead used when its accuracy is estimated to be better than offloading. To estimate the

Table 2.4. Ablation study on the contribution of offloading scheduling and local algorithm to depth estimation accuracy.

Scheduler	AbsRel ↓	Imp. over RR
RR	0.192	-
RR w/ LA	0.180	6.3%
AccuMO w/o LA	0.183	4.7%
AccuMO	0.175	8.9%

contribution of the two factors, we perform an ablation study where we run AccuMO without the depth estimation local algorithm (AccuMO w/o LA), and run RR additionally with the depth estimation local algorithm and the accuracy model (RR w/ LA), so that the scheduler switches to the local algorithm based on the accuracy estimates. Table 2.4 shows that RR w/ LA and AccuMO w/o LA each contribute about 60% and 40% of the total accuracy improvements, indicating that both are needed to attain high accuracy improvements.

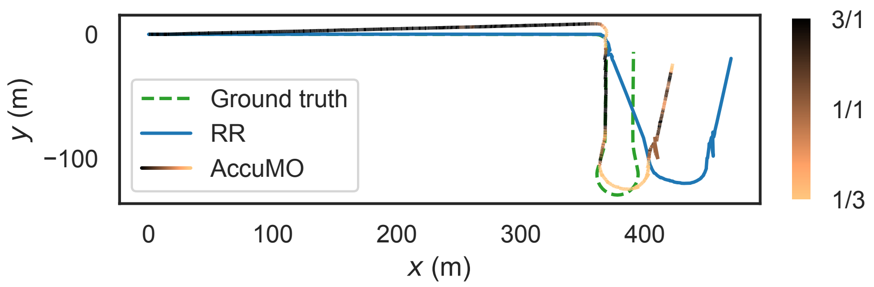


Figure 2.16. Odometry trajectories of a sample video. AccuMO's offloading schedule is overlayed on its trajectory.

Odometry. Since the odometry accuracy metric is calculated with segments, rather than frames, as the basic unit, in Figure 2.15 we calculate the accuracy of each frame by averaging over the accuracies of all segments containing the frame (see §2.6.2). We notice that for many frames, *e.g.,* frames 1800–2000, the per-frame accuracy is better than under RR even though the task is offloaded less frequently around these frames. This is because with high-accuracy-drop-rate frames (*e.g.,* frames 1400–1600) offloaded more often, the accuracies of all segments containing those frames are improved, which in turn improves the accuracies of less frequently offloaded frames (*e.g.,* frames 1800–2000) contained in these segments. As another example, in Figure 2.16, the RR trajectory mainly deviates from the ground-truth at places where the camera rotates. AccuMO offloads the odometry task more often at these frames, resulting in a trajectory that is closer to the ground truth. The extended impact of these high-accuracy-drop-rate frames also supports our accuracy model design of estimating the accuracy drop upper bound (§2.6.2).

2.8.6 Impact of Network and Server Configuration

Table 2.5. Task accuracies under different network and server configurations and scheduling algorithms on H/H videos.

	RR		AccuMO	
	Depth ↓	Odom ↓	Depth ↓	Odom ↓
802.11ac + 2080 Ti	0.239	0.125	0.216	0.083
5G + 2080 Ti	0.252	0.181	0.228	0.162
802.11ac + A40	0.217	0.105	0.200	0.078

In Table 2.5, we analyze the performance of our framework under the emulated 5G mmWave network. Furthermore, we also evaluate our framework against a more powerful server GPU — NVIDIA A40. The accuracies of both tasks become worse when switching from 802.11ac to 5G, mainly due to the longer transmission time caused by the higher RTT (14 ms vs. 3 ms for 802.11ac). On the other hand, the task accuracies improve when switching to the A40 server GPU due to faster DNN inference (from 50.39 ms to 39.66 ms for depth estimation and from 53.73 ms to 50.56 ms for odometry). Under all configuration combinations, our framework consistently improves over RR by 7.8%–9.6% for depth estimation and 10.5%–33.6% for odometry, indicating that our framework is generalizable to different network and server configurations.

2.9 Summary

In this chapter, we presented to our knowledge the first framework that dynamically schedules offloading of multiple compute-intensive DNN tasks of an AR app from a mobile device while optimizing the overall DNN inference accuracy across the tasks. We designed our framework to easily support diverse tasks by employing a general, two-level control feedback loop that adapts between alternative local execution options within each task module and globally balancing offloading among multiple tasks using MPC. Our evaluation results show that our framework can improve the overall task accuracy by on average 7.6%–14.3% over the best baseline under different accuracy weight ratios for depth estimation and odometry in edge-assisted AR.

3. ARISE: HIGH-CAPACITY AR OFFLOADING INFERENCE SERVING VIA PROACTIVE SCHEDULING

This chapter is based on work published in *Proceedings of the 22nd ACM International Conference on Mobile Systems, Applications, and Services* [84], https://doi.org/10.1145/3643832.3661894.

3.1 Introduction

3.1.1 Motivation

To deliver a truly immersive and interactive experience to users, Augmented Reality (AR) and Mixed Reality (MR) apps need to perform a number of challenging computer vision tasks to understand and interact with the physical environment, such as object detection [9], [16], [31], depth estimation [10], [85], and odometry [86], at acceptable accuracy. Furthermore, all of the tasks in the AR app performed on each frame are latency-critical; the results for the current frame need to be available in the *current frame interval*, *e.g.,* 16.7 ms at 60 FPS [9], as otherwise they will miss the rendering task for the current frame.

Due to their high accuracy, Deep Neural Network (DNN) models have been increasingly used to support these complex AR tasks (*e.g.,* [2]–[7]). However, running state-of-the-art DNN models on commodity mobile devices could take hundreds of milliseconds or even seconds [9]–[11]. For example, it takes 254 ms to run the DenseDepth [87] model on a Pixel 7 phone (on the mobile GPU with TensorFlow Lite). While lightweight models like FastDepth [63] and MobileNetV3-SSD [88] (for depth estimation and object detection respectively) are capable of running on-device in real-time, they fall short in accuracy; they achieve 0.35 AbsRel and 0.57 IoU on the datasets used in our evaluation (§3.6.2), compared to server-grade models such as DenseDepth [87] and ByteTrack [89], which achieve 0.12 AbsRel (lower is better) and 0.90 IoU (higher is better). To this end, offloading, also known as edge-assisted design, has been proposed [9], [13], [14], [90], where camera frames are uploaded to a cloud or edge GPU server for faster DNN inference.

To actually deploy AR apps based on edge-assisted AR design requires effective edge server support for serving potentially a large number of concurrent offloading requests of AR tasks from many mobile clients. Instead of maintaining its own servers, a cost-effective way is for an AR app vendor to use commercial Machine-Learning-as-a-Service (MLaaS) [23], [24], *e.g.,* deployed in the edge cloud (*e.g.,* AWS Wavelength [15], where edge servers are located in the same city and connected to the 5G infrastructure via low-latency links) to serve the AR clients in the vicinity. To effectively manage cost, MLaaS services are usually powered by DNN models under the control of service providers, *e.g.,* one or a few models for each AR task [91], [92]. A primary business objective of an MLaaS provider is to increase the capacity of the cluster (*i.e.,* the number of AR clients that it can serve concurrently), or provision the cluster size for a given user base in order to maximize its cost-effectiveness.

In such a shared edge-cloud setting, the AR apps running on the mobile clients often exhibit diversity in task accuracy requirements. For example, healthcare apps [25] require high accuracy while tourism apps can get by with moderate accuracy [26]. In such a deployment, the vendor or the users of the AR apps specify an accuracy SLA for each AR task (*e.g.,* object detection) that is required to ensure satisfactory app QoE, and the MLaaS provider tries to maximize the capacity of the GPU cluster in concurrently serving multiple AR clients. In this chapter, we focus on optimizing the capacity of individual servers, which is a key challenge and building block in increasing the capacity of a cluster.[1] We formally state the *AR offloading inference serving* problem as: *Given an edge/cloud server cluster serving offloaded inference tasks from AR clients, how to maximize the number of clients supported by individual servers while meeting the per-client AR task accuracy SLAs?*

AR apps often perform and need to offload multiple types of tasks. An efficient way of serving multiple types of AR tasks for a large client base is to divide up the servers in the cluster, such that each server serves one type of AR tasks, since such specialization unlocks more batching opportunities, as shown in previous serving systems [28], [93]. We follow the same design principle and each server only serves one type of AR task.

Difference from video analytics serving. Video analytics serving (*e.g.,* [14], [30], [49], [94]–[101]) is similar to AR offloading serving in that one or multiple clients offload a stream of requests to the server for inference. However, there is a subtle but fundamental difference

[1]↑We leave cluster-wide optimizations such as load balancing and dynamic scaling as future work.

between AR offloading serving and video analytics serving: AR apps desire task results for every frame in order to render virtual objects per frame. When the E2E offloading latency of an offloaded frame is longer than a frame interval due to server inference and frame transfer delay, the AR client resorts to local tracking (detailed in §3.2) to derive result for the current frame based on the last server-returned result. As a result, the E2E offloading latency directly affects the staleness of the server-returned result which impacts the AR task accuracy for all the frames. *Most of the video analytics systems do not factor in the impact of E2E offloading latency on app accuracy.* While a few works take it into consideration [38], [41], they only support serving a single client.

Difference from generic inferences serving. Compared to generic inference serving (*e.g.,* [28], [36], [93], [102]–[108]), AR inference serving faces a significant new complication. Generic inference serving systems assume DNN inference requests are independent, each with its own latency deadline or accuracy constraint. In contrast, in AR offloading serving, the stream of continuous requests offloaded from each AR client are *not unrelated*: due to the use of local tracking, for each client, *the choice of which frame to offload (offloading frequency and timing) and E2E offloading latency of the offloaded inference directly affect the AR task accuracy of all the frames of that client.*

3.1.2 Our Contributions

In this chapter, we present a framework that tackles this important AR inference serving problem. We observe that the key design optimization in generic inference serving systems — batched inference — is only exploited *opportunistically*, since such systems assume the inference requests are independent and have no control over request arrivals. The above key difference of AR offloading serving, namely, the AR task accuracy for consecutive frames is also affected by the offloading frequency and E2E delay, also presents a unique design opportunity — *coordinating client offloading schedules to work synergistically with batched inference on the server to maximize the effectiveness of batched server inference while meeting per-client accuracy SLAs.* While some video analytics serving systems also control clients' offloading schedules [30], [95], [98], [100], they do not coordinate client schedules with server inference; coordinating client schedules can boost the effectiveness of batching, as requests within a batch are from different clients.

Exploiting the above design opportunity, i.e., coordinating AR client offloading schedules to maximize the batched server inference, however, faces several challenges: (1) there exists complex relationship between the control knob values (client offloading schedule and server batch size) and the accuracy of *a single client*; (2) there exists interference among the effects of the control knobs *across clients*, as the schedule for any client affects server batching and hence E2E offloading latency (and accuracy) for other clients; (3) modeling per-client accuracy, a pre-requisite for online scheduling, is hard as it is not only affected by the per-client offloading schedule and server batch size, but also frame content, which can change dynamically during the execution of an AR app.

Our proposed AR offloading inference framework ARISE (AR Inference Serving Engine) untangles the intricate interplay among the control knobs across clients via a centralized but scalable scheduler that *proactively* coordinates the offloading schedules of AR clients and the batched inference on the edge server. We first develop a novel lightweight, online accuracy estimator that estimates the AR task accuracy for the current frame for each AR client under different offloading frequency, E2E latency, and dynamically changing frame content. We then design a novel scheduler that decouples deriving per-client offloading schedules and server batching schedule in two steps: (1) it first calculates a pseudo-optimal offloading frequency per-client leveraging the accuracy estimator; (2) it then greedily packs future offloaded requests from all clients into fewer large batches and coordinates client requests accordingly without violating per-client accuracy requirements.

We implemented the ARISE framework on commodity Android phones and GPU servers. Our evaluation using a large set of emulated AR clients and a small-scale testbed of 10 phones show that compared to Clipper-like [103] and Chameleon-like [95] systems, ARISE provides significantly higher capacity in serving AR clients, by up to 5.2x and 6.9x for the depth estimation task and by 1.9x and 2.0x for object detection on an NVIDIA A40 GPU server.

In summary, our main contributions are as follows:

- We present, to our best knowledge, the first framework for serving concurrent edge-assisted AR clients that maximizes the serving capacity of a server while satisfying the accuracy SLAs of originating AR apps.
- We present an AR inference serving scheduler that proactively coordinates offloading request streams from AR clients to maximize server batching opportunities.
- We present a lightweight AR task accuracy estimator under the commonly used offloading+local tracking based edge-assisted design (§3.2).
- We implement and experimentally validate our ARISE framework design by comparing it with various baselines including Clipper-like and Chameleon-like systems for two representative AR tasks.

3.2 Background: AR Task Accuracy under Edge-Assisted Design

3.2.1 The Offloading + Local Tracking Paradigm

In edge-assisted AR, even with powerful GPUs, typical DNN inferences still take tens of milliseconds, failing to return the result within the same frame interval. For example, models in Meta's object detection model zoo [37] have a median inference time of 52.5 ms on Tesla V100, much longer than the of 16.7 ms frame interval needed by AR apps running at 60 FPS [9], and the result of an offloaded frame may come back several frame intervals later.

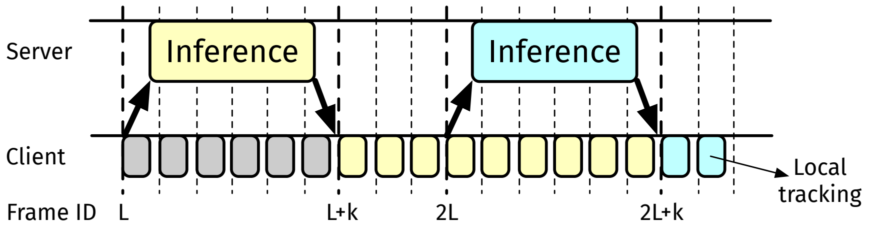


Figure 3.1. The offloading+local tracking paradigm, with offloading interval of L, and E2E offloading latency of k frame intervals. Frames $L+k$ to $2L+k-1$ reuse the inference result for frame L through local tracking.

Instead of simply using the last server-return result for the current frame, which was for the last offloaded frame, recent edge-assisted designs [9], [10], [16], [26], [29], [31], [38]–[45]

have adopted the local tracking technique to generate more accurate results for AR tasks including object detection, human pose estimation, odometry, and more. Specifically, a local tracker runs on mobile device and adjusts the DNN inference results for the last offloaded frame f_l sent back by the server to generate refined results for the current frame f_c, by analyzing the changes between the stale frame f_l and the current frame f_c, as shown in Figure 3.1. Such local trackers are fast and can typically finish in a fraction of the current frame interval. Local trackers are task-specific and often custom-designed for each type of tasks (*e.g.*, [9], [44]).

3.2.2 Impact of Tracking Stride on Accuracy

While local tracking improves the accuracy of the result for the current frame f_c (compared to directly reusing the last server-returned result), the gap between its accuracy and that of running the server DNN model directly on f_c (if we could) still widens with *tracking stride*, defined as the frame distance between f_l and f_c, due to increased staleness of the results for frame f_l. For example, in Figure 3.1, the client offloads every L-th frame, and E2E offloading latency is k. The result for frame L (offloaded at frame interval L) will return at frame interval L+k-1, and be used by local tracking to generate tracked results for frames L+k, . . . , 2L+k-1, which would have tracking strides of k (min), k+1, . . . , L+k-1 (max), respectively.

In scenarios where AR task accuracy can be met with less frequent offloading, whether in single-client scenarios (e.g., [9], [16], [38]) or in multi-client scenarios (this chapter), reduced offloading frequency (L) saves server and network resources used by the client. Furthermore, in multi-client scenarios, employing batched inference leads to variable E2E offloading latency (k) that is dependent on the batch size. This variability in offloading frequency and E2E latency has three immediate implications on AR task accuracy:

O1: Higher offloading frequency (smaller L) improves AR task accuracy. Offloading more frequently reduces the number of times a stale result is used (L), which improves tracking accuracy.

O2: Lower E2E offloading latency (smaller k) improves AR task accuracy. Lower E2E offloading latency reduces the staleness of last server-returned result used in tracking, which also improves tracking accuracy.

O3: The same accuracy SLA can be achieved by trading off offloading frequency (L) with E2E offloading latency (k). It follows from **O1** and **O2** that AR task accuracy can be improved by reducing either the offloading frequency (L) or the E2E offloading latency (k).

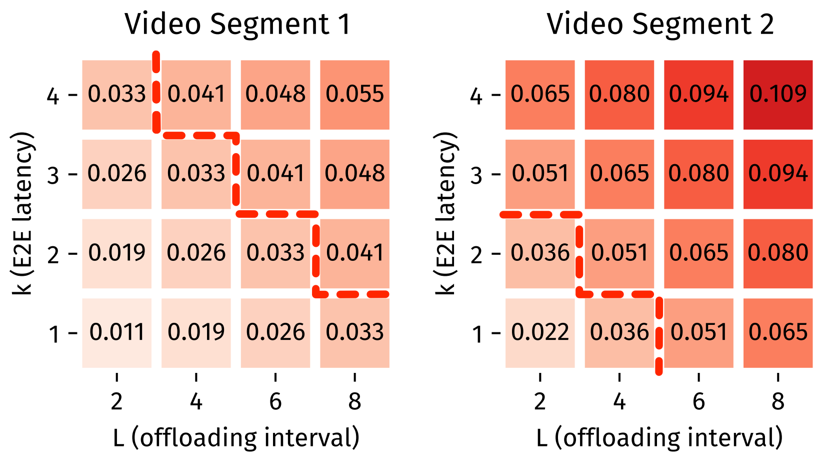


Figure 3.2. Accuracy drop (our accuracy SLA metric) of offloading the depth estimation task relative to running the DNN model directly, under different (L, k) combinations for two 2-second video segments. With an accuracy SLA of 0.040, only combinations below the red line satisfy the accuracy SLA.

Accuracy measurements in Figure 3.2 corroborates our observations. We control the offloading of two 2-second video segments (60 FPS) under different offloading frequencies and E2E offloading latencies, and the server runs the DenseDepth [87] model for the depth estimation task. The AR task accuracy improves with lower offloading interval (**O1**) and lower E2E offloading latency (**O2**). It also shows that the same accuracy SLA can be achieved by trading off the two (**O3**). However, the range of satisfactory L and k varies with different video segments (contents), indicating the need for an accuracy estimator that takes frame content into account (§3.4.3).

3.3 Design Opportunities and Challenges

3.3.1 Design Opportunities

Main idea. Generic inference serving systems maximize the server throughput while satisfying the latency or accuracy constraints of individual requests. To this end, such systems employ adaptive batched inference as a main optimization technique. However, batched inference in such systems is only exploited *opportunistically*, since such systems assume the inference requests are independent and have no control over request arrivals. An AR client, on the other hand, offloads a stream of requests, and the AR task accuracy for consecutive frames is affected by both the offloading frequency and E2E delay. this key difference presents a unique design opportunity: *the client offloading schedules can be coordinated to work synergistically with batched inference on the server to maximize the effectiveness of batched server inference while meeting per-client AR task accuracy SLAs.*

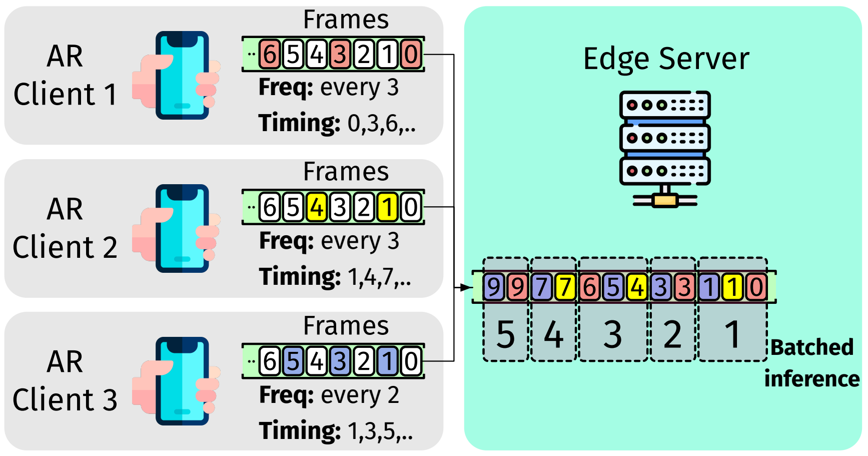


Figure 3.3. Control knobs in AR offloading serving, including (1) offloading frequency and (2) offloading timing for each client, and the (3) batch size for batched inference on server.

Control knobs. In generic inference serving, adaptive batched inference is achieved by dynamically tuning one important control knob:

- **Server batch size:** It controls the number of requests to group together and perform DNN inference in a single shot. A larger batch size improves GPU efficiency and thus server capacity. The downside of a larger batch size is longer inference latency, which could negatively impact the application performance or accuracy.

The adaptive batched inference technique also applies to the AR offloading inference problem, where the choice of batch size affects the E2E offloading latency (k).

Additionally, the offloading schedule of individual AR clients can be dynamically adjusted via two client-side knobs (Figure 3.3):

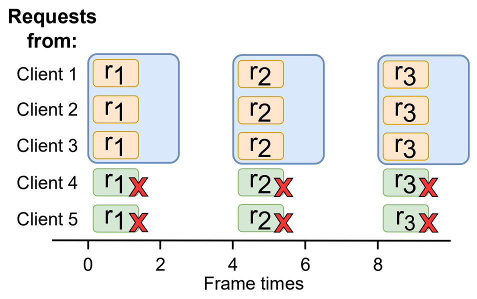
- **Offloading frequency:** It determines *how many* frames each client offloads in a period of time (L), *e.g.,* the 3 clients in Figure 3.3 offload every 3rd, 3rd, and 2nd frames, respectively. Higher offloading frequency improves AR task accuracy (**O1**) and allows for relaxed E2E latency and hence a larger server batch size for a given accuracy SLA (**O3**).
- **Offloading timing:** It determines *which* (and hence *when*) frames are offloaded, *e.g.,* client 1 offloads its frames 0, 3, 6, etc., whereas client 2 offloads frames 1, 4, 7, etc.. As we will see below, tuning this knob impacts the grouping of requests into batches, and thus the server capacity.

An example. Figure 3.4 gives an example on how these additional knobs help unlock more batched inference opportunities and hence improve the server capacity. Under a given accuracy SLA, AR clients can trade off offloading frequency (L) with E2E offloading latency (k) (**O3**). For example, we assume clients 1–3 can either offload once every 8 frames, which allows an E2E latency of 2 frame times (resulting in batch sizes up to 2), or once every 4 frames which allows an E2E latency of 2.5 (batch size 3), while clients 4 and 5 have different tradeoffs due to content difference (§3.2).

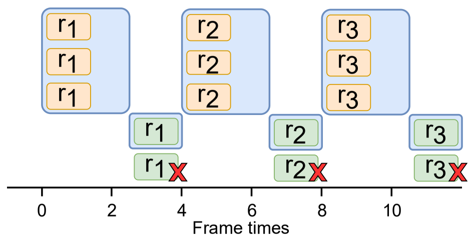
Figure 3.4a shows one possible trace of generic inference serving where all clients offload requests every 4 frame times. While 5 requests arrive at the sever at the same time, the system can only process at most 3 (restricted by the first 3 clients' E2E offloading latency limit) in one batch, which will finish at time 2.5. By this point it is too late to process the other 2 requests, as they require a maximum E2E latency of 1.5 frame times but have already been queued for 2.5 frame times. As a result, requests from clients 4 and 5 are dropped and the server is unused from time 2.5 to 4.0.

Feasible (k, L) Pairs			
Client 1, 2, 3		Client 4, 5	
k	L	k	L
2.0 2.5	8 4	1.5 2.0	6 4

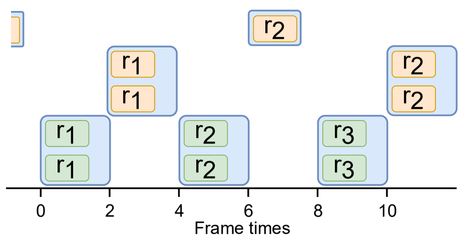
Inference Latency	
Batch Size	Latency
1 2 3	1.5 2 2.5



(a) Generic inference serving.



(b) Controlling batching and offloading timing.



(c) Controlling batching, offloading timing and frequency.

Figure 3.4. Controlling offloading schedules (timing and frequency) of AR clients unlocks more batched inference opportunities and enables server to support more clients. The top-left table lists for each client the feasible choices of E2E offloading latency (k) and offloading interval (L) pairs due to accuracy SLA. The bottom-left table gives the maximum batch sizes that can be tolerated by different E2E offloading latencies (k). Each blue box corresponds to one batched inference on the server.

In contrast, in an AR inference serving system, the system can control request arrivals by adjusting each client's offloading timing and frequency. In Figure 3.4b, by shifting the first request of client 4 to arrive at time 2.5, the server can additionally perform an inference of batch size 1, from time 2.5 to 4.0, increasing server capacity to support 4 clients.

Further, by controlling both request arrival timing and offloading frequency, the server can support all 5 clients, as shown in Figure 3.4c. By reducing the offloading frequencies of clients 1–3 to once every 8 frames, although this will result in requiring a tighter E2E latency (of 2 frame times) and thus a smaller batch size of 2 for these clients, the request arrival timing of the clients can be coordinated such that requests from all 5 clients can be served under their batch size constraints (and hence their accuracy SLAs).

3.3.2 Design Challenges

Exploiting the above new design opportunity, *i.e.,* jointly tuning per-client offloading frequency and timing and server batching (control knobs) to maximize the server capacity (objective) while meeting the per-client accuracy SLAs, however, faces several challenges:

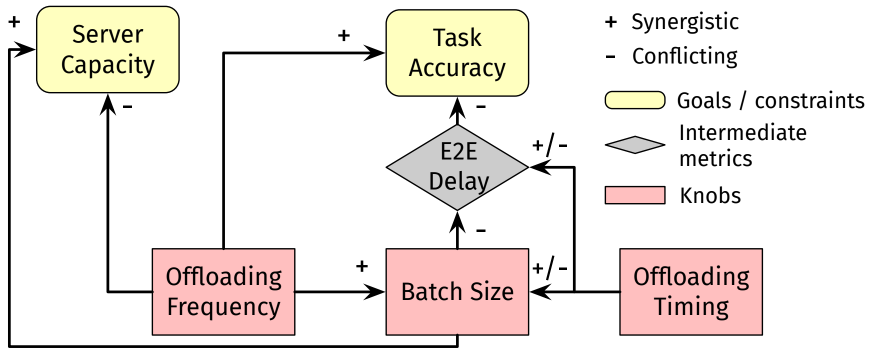


Figure 3.5. Relationships between control knobs and design goals in AR inference serving.

C1: Complex relationship between control knob values and design goals for a single client. For each client, tuning the offloading frequency and batch size can affect the server capacity and task accuracy in complex ways, as shown in Figure 3.5. (1) Increasing the *offloading frequency* improves the task accuracy (**O1**) and the chance of building larger batches which indirectly increases server capacity, but (directly) reduces the server capacity as each client imposes more load on the server. (2) Increasing the *batch size* directly improves the server capacity, but incurs higher queuing and batch inference time, which in turn results in higher per-request E2E delay and thus lower accuracy (**O2**). (3) The reduced accuracy from larger batch size can be compensated by higher offloading frequency (**O3**), which in turn reduces the server capacity. The amount of compensation depends on the relationship between accuracy, E2E delay, and offloading frequency. It is challenging to determine the optimal balance between batch size and offloading frequency.

C2: Intricate interplay among the control knobs across clients. Further, finding the combinations of control knobs for a set of clients becomes even more challenging as the choices for multiple clients can interfere with each other in complex ways. (1) The latency of a chosen batch size that is sufficient for some clients (*e.g.,* with lower accuracy SLAs) may be too long for others (*e.g.,* with higher accuracy SLAs). (2) Under local tracking based AR offloading (§3.2), interference among offloading requests by different clients, due to incompatible offloading timings, can further elongate the E2E offloading delay and hence lower the client accuracy (Figure 3.5). Consider the server is serving 10 clients and the minimum batch sizes calculated for the 10 clients in isolation is 5. If the requests from the 10 clients happen to arrive at the same time, the server has to either select a batch size of 5, which meets the accuracy target of the most stringent client, but postpones the rest 5 requests to the next batch, causing *batch-level* queuing delay and violation of accuracy constraint for those clients, or it selects a larger batch size which violates the accuracy target of the most stringent client. In an alternative scenario, if the requests from the 10 clients are evenly spaced out in time, creating a batch of 5 requests will require the first arrival request to wait for 4 more request arrival, causing *intra-batch* queuing delay.

3.4 ARISE Design

3.4.1 Design Rationale

ARISE schedules client offloading requests and server batched inferences to maximize the server capacity while satisfying the accuracy SLAs of all clients. One possible approach is letting each client decide on its own offloading schedule in a distributed manner. However, such a distributed approach makes it hard to tackle the intricate interplay among the control knobs across the clients (**C2**). For example, a client experiencing longer E2E request delay may react by increasing its offloading frequency to meet the accuracy SLA, which can lead to higher load on the server and even longer E2E delay. To this end, we propose a centralized scheduler on the server side that proactively coordinates requests from all clients and optimizes server capacity. We also design the scheduler's complexity to be linear in the number of requests, which ensures the scheduler is scalable to support a large number of clients. On the other hand, to ensure all clients meet their accuracy SLAs, an accuracy estimator is needed for each client to work in tandem with the scheduler. We design a lightweight accuracy estimator that runs on each client and sends the accuracy estimates to the server. With the accuracy estimates, the scheduler is able to calculate the accuracy of all clients in a specific scheduling plan and ensure all clients meet their accuracy SLAs.

3.4.2 Architecture Overview

Figure 3.6 shows the workflow of ARISE. We envision ARISE will be deployed on the servers in a cluster, where a load balancer is responsible for directing new clients to the servers and help migrate extra clients that exceed the capacity of a server. Each server's scheduler determines whether to take up more clients or to remove clients depending on the resource requirements of the current clients it is serving (evicted clients are redirected by the load balancer to other servers). As we will see shortly, frequent scheduler invocation (every 200 ms) allows ARISE to respond to client arrival and departure promptly.

The rest of Figure 3.6 shows the workflow between the server and one of the served clients. The server consists of a DNN inference engine, a proactive scheduler, a data store for the schedule generated by the scheduler, and a data store for storing client states, including client task accuracy estimates and the timing of recently processed batches and frames. The client is equipped with a camera capturing frames in real time, *e.g.,* at 60 FPS, a stored copy of the latest schedule that dictates the frames to offload, a local tracker, and the accuracy estimator.

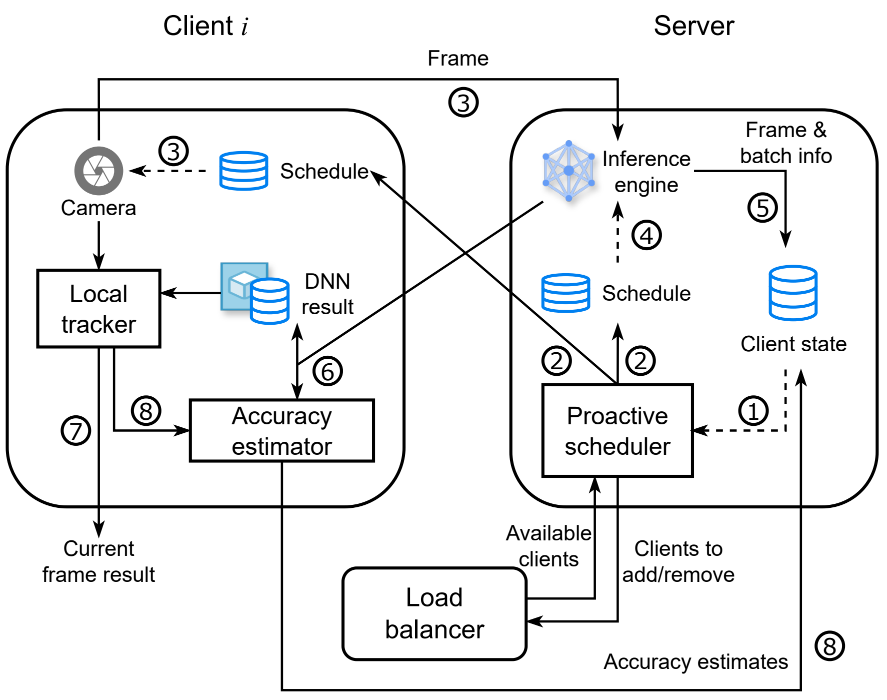


Figure 3.6. The workflow of ARISE with one of the served clients. The proactive scheduler coordinates the offloading schedules across clients and with batched inference. The accuracy estimator estimates task accuracy by exploiting unique properties of local tracking.

Periodically (every 200 ms), the scheduler generates a schedule for the near future, *e.g.,* 1 s, based on the latest client states ❶. The schedule contains IDs of the frames to be offloaded by each client and the grouping of the frames from multiple clients into batches, and is both stored on the server and sent to the clients ❷. The client offloads selected frames ❸, and the server performs batched inference of client requests ❹, both as dictated by the schedule. After inference, the server updates the client state (timing of both the batch and frames in it) for future scheduling ❺, and it sends the inference results back to each client for both local tracking and accuracy estimation ❻. The local tracker is executed for every frame using the latest DNN results and generates result for the current frame for use by upper level AR applications ❼. Finally, the accuracy estimator of each client calculates the current accuracy estimates based on DNN results and local tracking results for the same frame (the one with the last server-returned result), and updates the client states for future scheduling ❽.

3.4.3 Accuracy Estimation

ARISE expects each client to specify its accuracy SLA according to their application needs. In practice, the accuracy is upper-bounded by what can be achieved in an idealistic offloading scenario where each user is given a dedicated GPU server. Therefore, it is more meaningful for ARISE to meet accuracy drop targets (*e.g.,* within each time window) relative to this idealistic scenario, which we refer to as the *reference setup*. We envision ARISE users will specify their accuracy SLAs as the accuracy drop from the reference setup.

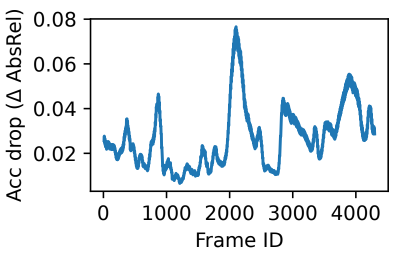


Figure 3.7. The accuracy drop (relative to reference setup) varies across frames.

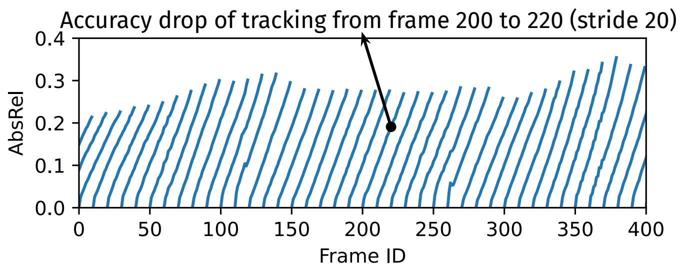


Figure 3.8. The local tracker accuracy drop (relative to DNN inference) under different tracking strides (from 1 to 30).

Challenges. As suggested in **O1**, **O2**, and **O3** (§3.2), a client's task accuracy is directly affected by the offloading frequency and per-request E2E latency. In addition, even under the same offloading frequency and E2E latency, the local tracker's accuracy may vary over time due to changing frame contents. Figure 3.7 shows the accuracy drop of offloading to a dedicated GPU server under batch size 1 and offloading interval 8 for depth estimation, with the task accuracy measured in absolute relative error (AbsRel). We observe that the accuracy drop varies significantly across frames (0.007–0.077). The detailed setup, including the dataset, DNN model, local tracker, GPU server, and network condition, can be found in §3.6.1.

Key insights. We make a key observation that all the factors affecting AR task accuracy mentioned above directly affect the staleness of the latest DNN result, which in turn affects the task accuracy, and thus a feature that captures the staleness of the DNN result (*i.e.,* staleness of the frame the result is for) could potentially bridge the gap between the affecting factors and resulting accuracy and be used to develop a lightweight accuracy estimator.

We observe that *tracking stride*, a unique feature in AR offloading (§3.2), is an ideal candidate to bridge accuracy estimation and impacting factors. First, tracking stride well captures the staleness of the DNN result. The relationship between the affecting factors and tracking stride is straightforward, as shown in observations **O1** and **O2**. Second, tracking stride directly correlates with the accuracy drop. In Figure 3.8, we plot for a sample video the accuracy drop compared to performing DNN inference on each frame individually. For each line, the x-intercept corresponds to the source frame of the local tracking, while each data point on the line corresponds to the accuracy of local tracking for a different destination frame. We observe that (1) The accuracy drop increases linearly with tracking stride, which enables us to estimate the accuracy drop rate (the slope) and multiply it with the tracking stride to get the accuracy drop under any tracking stride. (2) The accuracy drop rate exhibits temporal locality, *e.g.,* the slopes of the lines for frames between 150 and 250 stay the same. This allows as to approximate the accuracy drop rate of the current frame by estimating that of the last server-returned frame.

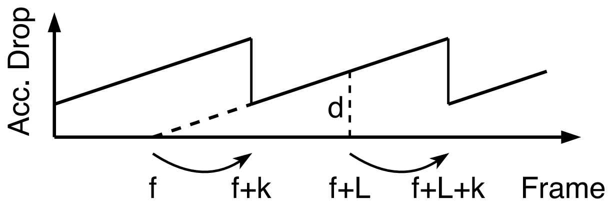


Figure 3.9. The estimation of accuracy drop d for frame $f + L$ is done by comparing two results for frame $f + L$, one obtained by local tracking on server-returned result for frame f, and one obtained by directly offloading frame $f + L$, whose result comes back at frame $f + L + k$.

Accuracy drop estimator. We start with estimating the accuracy drop rate, as illustrated in Figure 3.9. Assuming the E2E offloading latency is k frame intervals and the offloading interval is every L frames. Frame f is offloaded, and its result returns at $f + k$

and is used by the local tracker till frame $f + L + k - 1$ (including $f + L$). Similarly, frame $f + L$ is offloaded, and its result returns at $f + L + k$. At time $f + L + k$ the client holds both the DNN result and the local tracker result for $f + L$ (tracking result for frame $f + L$ is derived from DNN result for f), and it calculates the accuracy drop d between the two and divides it by the tracking stride $(f + L) - f = L$, which gives the slope f/L as the estimated accuracy drop rate.

We now use the accuracy drop rate to estimate the accuracy drop with regard to the reference setup. For both the schedule being profiled and the reference setup, we first estimate their accuracy drops with regard to offline DNN inference: we first calculate the tracking stride of each frame based on the offloading frequency (L) and E2E offloading latency (k), and then multiply the tracking strides with the accuracy drop rate to get the accuracy drops compared to offline DNN inference. Next, we calculate the accuracy drop difference between the schedule being profiled and the reference setup, which gives us the accuracy drop with regard to the reference setup.

3.4.4 Proactive Scheduling

Using the lightweight task accuracy estimator discussed above, the scheduler tries to maximize the number of supported clients while ensuring clients meet their accuracy SLA (expressed as accuracy drop thresholds) by dynamically adjusting the control knobs. However, the intricate interplay among the knobs makes it complicated to directly derive the optimal settings of all control knobs at once. To this end, we first decouple the knobs and generate offloading schedule per-client and serving batching plan in two steps. In Step 1, we derive the pseudo-optimal offloading frequency and batch size for each client, since their relationship with server capacity and task accuracy can be expressed in closed form. In Step 2, we “fine-tune” the generated schedule by greedily packing requests into larger batches and proactively adjusting the request arrivals according to the batch schedule to coordinate client requests and further improve the server capacity.

Step 1: Pseudo-Optimal Offloading Settings

The scheduler first calculates the optimal offloading settings — offloading interval and batch size — for individual clients assuming no queuing delay. Even so, the optimal settings are hard to compute as the optimal settings for a client need to be determined collectively by considering the states of other clients as well. This is because the accuracy for a client depends on both the client's offloading interval and server batch size, while the optimal batch size in turn depends on the optimal settings of other clients in the same batch, *i.e.,* a cyclic dependency. To this end, we propose a heuristic where we first calculate the optimal settings for each client i assuming that all clients have the same accuracy drop rate and SLA as the client of interest. Under this assumption, all clients share the same optimal settings, which can be obtained by solving the following equation:

$$\max_{\mathrm{oi},\mathrm{bs}} \frac{\mathrm{bs}}{\mathrm{lat}_{\mathrm{bs}}} \Big/ \frac{\mathrm{fps}}{\mathrm{oi}} \quad \text{s.t.} \quad \frac{1}{\mathrm{oi}} \sum_{s=\mathrm{e2e}_{\mathrm{bs}}}^{\mathrm{e2e}_{\mathrm{bs}}+\mathrm{oi}-1} \mathrm{dacc}_s - \mathrm{dacc}_{\mathrm{ref}} \leq \theta$$

where oi, bs, and θ represent the offloading interval, batch size, and accuracy SLA respectively, $\mathrm{lat}_{\mathrm{bs}}$ and $\mathrm{e2e}_{\mathrm{bs}}$ are the inference latency (in seconds) and the end-to-end offloading latency (in frame intervals) under a specific batch size, dacc_s is the accuracy drop under tracking stride s, and $\mathrm{dacc}_{\mathrm{ref}}$ is the accuracy drop under the reference setup.

The above equation maximizes the number of supported clients (pretending they are identical to client i), which equals to the server inference throughput ($\frac{\mathrm{bs}}{\mathrm{lat}_{\mathrm{bs}}}$) divided by the offloading frequency of each client ($\frac{\mathrm{fps}}{\mathrm{oi}}$), subject to the accuracy SLA, which is calculated as the average accuracy drop across all the frames that rely on tracking the offloaded frame, *i.e.,* with tracking strides $\mathrm{e2e}_{\mathrm{bs}}$ to $\mathrm{e2e}_{\mathrm{bs}}$+oi-1.

Step 2: Greedy Request Packing and Coordination

The optimal schedule calculated per client above assumes no queuing delay and perfect batching. In practice, offloading requests of uncoordinated clients may arrive at the server at any time, disrupting the above schedule. In Step 2, we perform *greedy request packing and coordination* that explicitly coordinates the timing of client request arrivals and batch formation. The algorithm consists of four steps: *simulate*, *adjust*, *verify*, and *apply* (Figure 3.10), which are described in detail below.

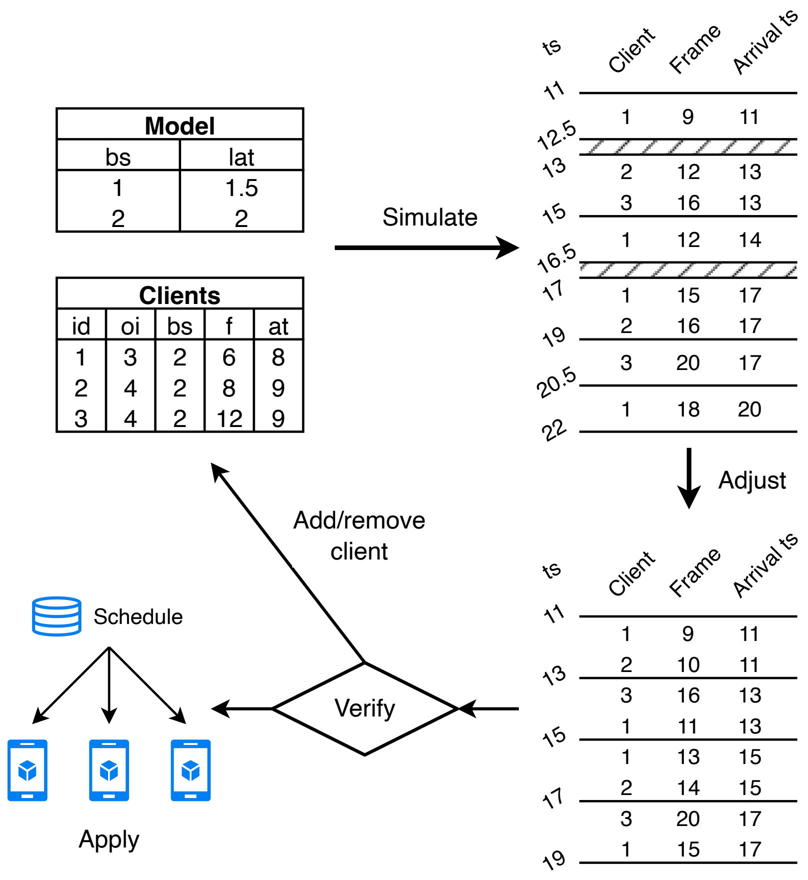


Figure 3.10. An example of greedy request packing and coordination. The "Model" table gives inference latencies of an example DNN model under different batch size. The "Clients" table gives the current states of the clients, including their ID, the selected offloading interval (oi) and batch size (bs), and the last offloaded frame (f) along with its arrival time (at). All latencies and timestamps are in frame intervals.

Simulate. Before packing and coordinating the client requests, we need to first simulate the request arrivals in the near future based on knob settings calculated in Step 1 and last frame's arrival time for each client. For example, in Figure 3.10, the last request of client 1 is frame 6, which arrives at time 8, and the offloading interval is 3. Thus, the next request will be frame 9 and should arrive by time 11. As a reference, in Figure 3.10, the requests are also grouped into batches similarly as how they would have been processed by a general inference serving framework, *i.e.,* opportunistic batching [103]. For example, at time 11, only 1 request is available, and thus a batch of size 1 is formed despite client 1 can tolerate a batch size of 2. On the other hand, at time 17, a batch of size 2 is formed despite 3 requests are available, since all the clients require a maximum batch size of 2.

Algorithm 2: Request packing and coordination (*adjust* step)

```
  input  : list of requests $R$ from simulate step in arrival order
           completion time $t_0$ of current batched inference
  output: scheduling plan $P$
1 $i = 0, t = t_0$;
2 while $i < R$.length do
      // pack
3     bs = $\text{argmax}_n\{r.\text{client.bs} \geq n, \forall r \in R[\text{i:i}+n]\}$;
4     batch = $R$[i:i+bs];
      // coordinate
5     for $r$ in batch do
6         $r$.arrival = $t$;
7         $r$.frameId = timeToFrameId($r$.client, $t$);
8     $P$.add(batch);
9     i += bs, $t$ += InferenceLatency(bs);
```

Adjust. As the core step of greedy packing and coordination, we adjust both the batches and the individual requests. Algorithm 2 shows the adjust algorithm that produces the adjusted scheduling plan. The outer loop (lines 2–11) goes through all the requests in the near future (generated in the *simulate* step) and packs them into batches; the inner loop (lines 5–8) goes through each request in the batch and coordinates them with server batched inference. The algorithm runs in linear time in terms of the number of requests being processed. To ensure the server resources are fully utilized, we regroup the requests into batches greedily without considering the arrival times of the requests (the batch size restrictions still apply). For example, in Figure 3.10, the first and the second requests are now grouped into the same batch that starts at time 11. Next, to resolve any conflicts between requests and minimize the queuing delay, we adjust the expected request arrival time and the frame ID of the requests according to the expected start time of the batches.

For example, in Figure 3.10, client 1 and client 2 requests originally arriving at time 17 are now moved forward to time 15, no longer conflicting with frame 20 from client 3. This adjustment step minimizes both batch-level and intra-batch queuing delays.

Verify. As the adjusted schedule affects offload timing and batching, which in turn affects task accuracy, we next verify the adjusted schedule whether all clients still meet their accuracy SLAs, by going through the adjusted schedule and calculating the accuracy drop of each client using the per-client accuracy estimator. If not, we remove one client from the server, and if yes, we add one more client from the list of available clients (along with their state when running on the previous server) provided by the cluster load balancer. We then repeat the simulate-adjust process until there is no need to remove a client and adding a client is not possible.

Apply. Finally, we apply the final schedule, which grants improved server capacity and meets the accuracy SLAs of all clients at the same time, by sending it to all clients. Both client offloading and server batched inference will follow the new schedule till the next scheduling iteration.

Practical issues. We perform several optimizations to ensure that the scheduler works smoothly in practice. (1) We impose a small overlap between the old and new scheduling plans, *i.e.,* the tail of the old plan is the same as the head of the new plan, which ensures clients can transition smoothly to the new plan without accuracy degradation. (2) Due to network bandwidth fluctuation, requests may not arrive at the exact time in the schedule. Our schedule sets aside a grace period, *e.g.,* 10% of the inference time for every batch to tolerate late requests, which strikes a balance between flexibility of the schedule and wasted server resource. (3) Network bandwidth fluctuation may also make estimating future request arrival time simply based on the last request unreliable. To this end, we perform regression based on recent request arrivals of a client and predict future request arrivals based on the regression model.

As different AR tasks are served by different servers (§4.1), they are typically scheduled independently for applications that rely on multiple AR tasks. In certain cases, *e.g.,* whether to execute one task depends conditionally on other tasks [28], [96], the scheduler, which executes at a fine granularity (200 ms), can leverage the temporal locality of task results (§3.4.3) and determine whether to run the task based on the most recent result of the dependency task.

3.4.5 Other Optimizations

ARISE automatically achieves *pipelining* between network transmission and server inference in the adjust step of proactive scheduling, where clients requests are regrouped to remove gaps between batches and offloaded frames are adjusted so that they arrive right before the batches begin execution. Furthermore, we perform JPEG encoding on both frames and DNN results (if applicable), which is efficient and has minimal impact on accuracy [109].

3.5 Implementation

We have implemented the ARISE server in about 2K lines of C++ code. We use TensorRT [110] as our DNN inference engine, and perform GPU-accelerated JPEG encoding and decoding using nvJPEG [111]. We implement two client implementations — an Android client and an emulated client. The Android client is implemented in a mix of Java and C++ and runs on the Android phone. The number of Android clients that can run is limited by the phones we have. To evaluate our system for a larger number of AR clients, we implemented an emulated client, written in Python, that emulates the behavior (including the computational latencies) of the Android client but runs on a server. In particular, the emulated client offloads the frames and receives the results like the real client, but emulates the local tracker latency and performs table lookup to get the local tracker accuracy, and hence is light-weight; a single server is able to host hundreds of emulated clients.

3.6 Evaluation

3.6.1 Evaluation Setup

Emulation. We run ARISE and other baselines on a server with an NVIDIA A40 GPU. We first evaluate our system with emulated clients. We emulate the mean and variance of dynamic 5G mmWave (1715 ± 57 Mbps downlink, 152 ± 6 Mbps uplink, 14 ± 2 ms RTT) and LTE (110 ± 17 Mbps downlink, 44 ± 8 Mbps uplink, 32 ± 5 ms RTT) network conditions [112] for each client using the tc tool (the clients' network conditions are independent). We simulate the client arrival following a Poisson process (on average 10 clients per second). Each client randomly selects a video from the dataset (see below) and replays frames in that video. Each experiment lasts 10 minutes. During the experiment, as the clients come and go, the server decides whether to add or remove clients depending on the clients' resource requirements. While in real-world deployment, a load balancer will help migrate the clients between servers in the cluster, for this experiment, we simply set up another server to serve the extra clients that exceed the capacity of the server being measured. The average number of concurrent clients served by the first server is measured and compared against the baselines.

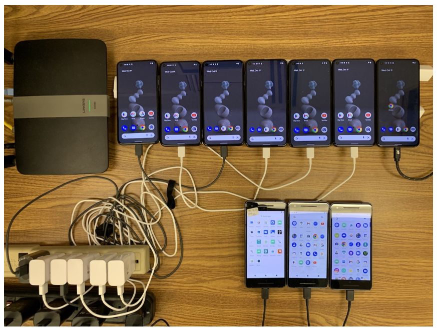


Figure 3.11. The testbed setup (server is not shown).

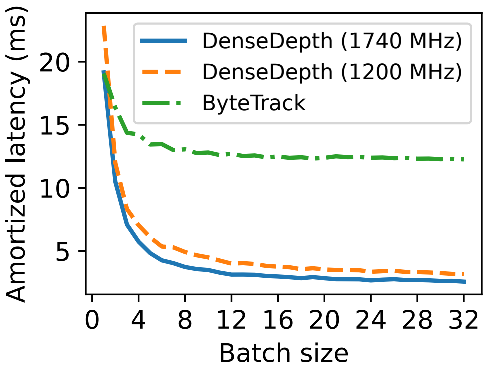


Figure 3.12. Amortized DNN inference latency.

Testbed. To verify the performance of ARISE against real mobile phones, we next evaluate our system on a small scale testbed setup (Figure 3.11) that consists of 10 smartphones (7 Google Pixel 5 and 3 Google Pixel 2). All phones are connected to an 802.11ac AP, which connects to the server through a 1 Gbps link. We still emulate the dynamic 5G mmWave network condition on top of it. Upon arrival, a client is assigned to one of the available phones and an instance of the Android client implementation is started on the phone. The phone will be free again when the client session finishes. An arriving client is rejected when all phones are busy. The rest of the setup are the same as above.

3.6.2 Evaluation Tasks

We test our system with two representative AR tasks separately: depth estimation and object detection. Due to the lack of AR-specific datasets, we follow the practice of recent AR systems [9], [29], [61] and use datasets collected for the target tasks.

Depth estimation. Depth estimation is an essential AR task that estimates the depth map — the distance of each pixel relative to the camera — given an RGB frame. We employ the popular DenseDepth [87] model. Figure 3.12 (1740 MHz) shows the amortized DNN inference latencies (batch inference latency divided by batch size) under different batch sizes. We use warping [113] as the local tracker. We use a dataset generated by CARLA [80], which contains 20 videos with resolution 448×128 (the input resolution of DenseDepth[2]), each lasting for 70 seconds at 60 FPS with diverse frame content and varying accuracy drop rates. We evaluate the accuracy of the depth maps using the absolute relative error (AbsRel, lower is better) [83].

Object detection. Object detection is another important AR task that helps AR devices to understand the semantics of the surrounding environment. We use ByteTrack [89], a video object detection model that provides smooth object trajectories compared to image object detection models, as our DNN model. Figure 3.12 shows the amortized DNN inference latencies. We use a Lucas-Kanade based local tracker that estimates new bounding box locations based on the optical flow [114]. We use videos from MOT17 [115] and evaluate object detection accuracy using Intersection over Union (IOU).

3.6.3 Baselines

We compare ARISE against the following baselines.

Static. This baseline uses the same offloading interval and batch size during the experiment and across all clients. The maximum number of clients it can support and the corresponding configuration values and are determined offline based on pilot experiments that search

[2]↑While AR device cameras usually have high resolution, the camera frames are typically down-sampled to match the input resolution of DNN models [9], [16], [29].

through the configurations, such that the averaged accuracy across all frames for each client meets its accuracy SLA. The clients are equipped with local trackers.

Clipper-like (dynamic batching, no dynamic offloading interval). DNN serving systems like Clipper [103] are not directly comparable to ARISE due to their focus on per-request accuracy (which is latency oblivious) or just latency. To this end, we implement a Clipper-like system that is enhanced with local tracking and lightweight accuracy estimation as in ARISE. It performs dynamic batching in the same way as in the simulate step in §4.5.5. However, all clients offload at a fixed offloading interval (the value is chosen offline in the same way as Static), since requests from the same client are treated independently by such DNN serving systems.

Chameleon-like (dynamic offloading interval, no dynamic batching). We also implement a system that resembles video analytics pipelines like Chameleon [95] but enhanced with local tracking. Chameleon uses profiling-based accuracy estimation which is too compute-intensive to scale, and thus we replace it with the lightweight, online accuracy estimation method in ARISE. The offloading interval of each client is dynamically determined based on the accuracy estimate. However, batching is not employed.

ARISE-α. This baseline takes advantage of most techniques employed by ARISE, including lightweight accuracy estimation, dynamic offloading interval, and dynamic batching. However, the server-side scheduling is done reactively, without the greedy packing in the *adjust* step in §4.5.5. Essentially, the offloading schedule is the output of the *simulate* step, governed by the *verify* step to ensure that the SLA is met.

Note that the first 3 baselines are "strong baselines" beyond practical algorithms because key offline parameters are chosen based on prior knowledge of the target workload.

Load adjustment for the baselines work as follows. For Clipper-like, Chameleon-like, and ARISE-α, clients are added or removed based on the presence of pending requests, *i.e.,* a fixed number of clients are removed when some requests are delayed till the next batch, which cause them to miss their accuracy SLA, while more clients are added if no requests are delayed within 400 ms. For the Static baseline, the number of clients to support is fixed, as discussed above.

3.6.4 Main Results

We first evaluate the systems with emulated clients and the A40 GPU on the depth estimation task.

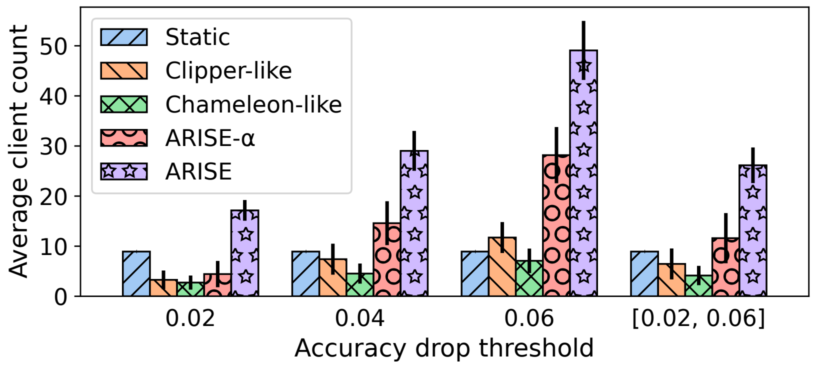


Figure 3.13. Average number of concurrent clients and the standard deviation under different accuracy SLAs.

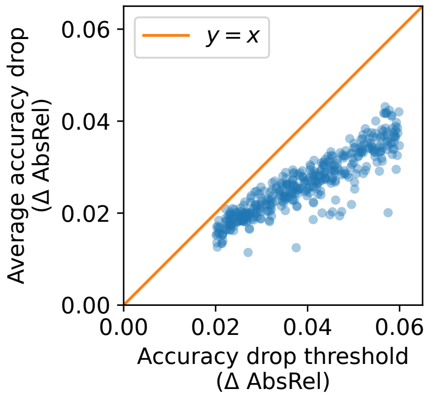


Figure 3.14. Average accuracy drops vs. accuracy SLAs for ARISE with SLAs in [0.02, 0.06].

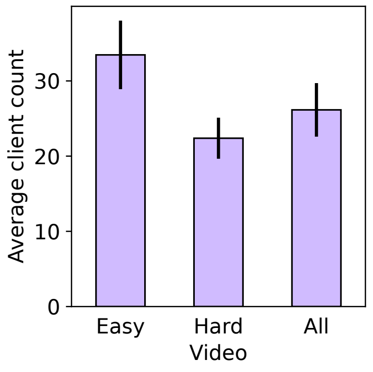


Figure 3.15. Average client count under different frame difficulties and accuracy SLAs in [0.02, 0.06].

First, Figure 3.13 compares the number of clients supported by different systems under 5G mmWave when all clients have the same accuracy SLAs 0.02, 0.04, 0.06 (SLA values are picked as a fraction of the typical range of the tasks accuracy metric), and when the SLAs are drawn randomly from a uniform distribution of range [0.02, 0.06]. In all experiments, Static is configured with fixed offloading interval 3 and batch size 5, while Clipper-like has a fixed offloading interval of 3. The average accuracy for all clients meet the SLAs. For example, in Figure 3.14, which plots the average accuracy drops vs. the accuracy SLA when ARISE serves clients with accuracy SLA drawn from [0.02, 0.06], all data points are below $y = x$, indicating that ARISE meets the accuracy SLAs of all clients.

We make the following observations about Figure 3.13. (1) ARISE improves over all baselines under all accuracy SLA choices by 1.7x–6.9x. ARISE improves over ARISE-α by 1.7x–3.9x, indicating the importance of proactive scheduling in resolving the conflicts between requests and increasing the number of clients supported by the server. (2) On the other hand, the improvement of ARISE-α over Clipper-like (by 1.8x–2.4x) and Chameleon-like (by 1.6x–4.0x) shows that tuning both dynamic offloading intervals and batching is important in improving serving performance. (3) Static performs better than other baselines such as ARISE-α under tighter accuracy SLAs. This is because Static's configuration values are selected so that the per-client average accuracy drops are with the SLA, while all other baselines instead strive to ensure that the clients meet their accuracy SLAs at *all times* (by adjusting the number of clients). Thus, while Static allows extra delays caused by uncoordinated requests as long as the per-client average accuracy drops are within the SLA, the dynamic baselines without proactive scheduling try to keep request conflicts minimal, which causes the server to be under-utilized, as we will discuss in §3.6.6. (4) The ratio of average client count between ARISE and other dynamic baselines (Clipper-like, Chameleon-like, and ARISE-α) becomes smaller as the accuracy SLA becomes larger, since the impact of request conflicts becomes smaller as clients can tolerate longer E2E offloading latencies.

Impact of frame content. To study the impact of frame content on ARISE performance, we partition the 20 videos in the CARLA dataset into two groups of 10 videos each, one with high accuracy drop rates (on average 0.007 per frame) and the other with low accuracy drop rates (on average 0.005 per frame). We next run ARISE with only high accuracy drop videos (hard clients), low accuracy drop videos (easy clients), and all videos together, respectively. Figure 3.15 shows the average number of clients (accuracy SLA [0.02, 0.06]) that ARISE can support. Compared to serving hard clients, ARISE can support 50% more easy ones. In either case, ARISE dynamically adjusts the number of supported clients to ensure all clients meet their accuracy SLAs.

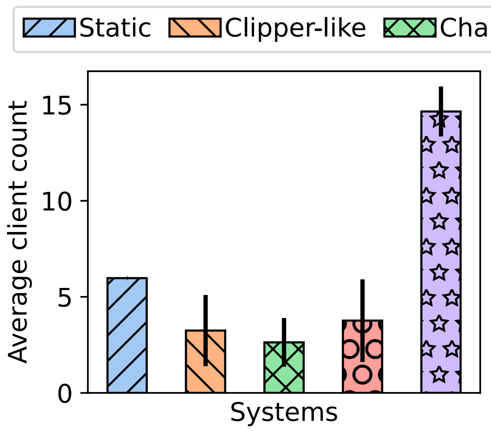


Figure 3.16. Average client count under LTE at accuracy SLA 0.02.

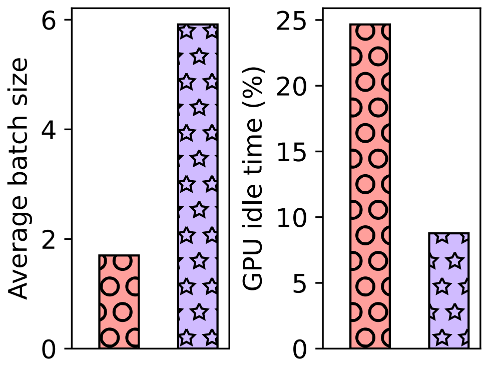
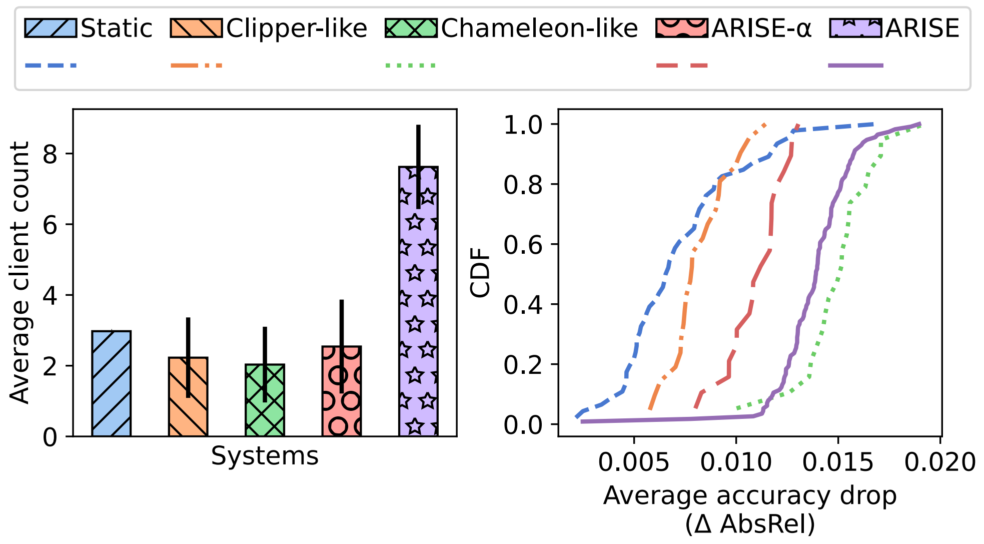


Figure 3.17. Average batch size and server idle time of ARISE-α vs. ARISE at accuracy SLA 0.02.

Impact of network condition. Figure 3.16 shows the average number of clients supported by each system under LTE network condition, which entails longer network latency and different network dynamics. In this experiment, Static is configured with offloading interval 3 and batch size 3 for all clients to meet the accuracy SLA (0.02), and Clipper-like uses fixed offloading interval 3. While all systems support fewer clients compared to running under 5G network condition, ARISE still significantly improves over other baselines and supports 2.5x–5.6x more clients, which demonstrates the robustness of ARISE under different network conditions.

3.6.5 Testbed Verification

To verify that our framework works on real clients, we next evaluate the systems on the small-scale testbed. As the number of clients supported by some of the baselines exceeds the number of phones we have, which makes it hard to compare between the systems, we restrict the GPU clock frequency to 1200 MHz, and Figure 3.12 shows the longer batch inference latency compared to no GPU clock frequency limit.



(a) Average number of concurrent clients. (b) CDF of per-client average accuracy drops.

Figure 3.18. Testbed results at accuracy SLA 0.02.

Figure 3.18 shows the number of clients supported and per-client accuracy drops for different systems under 5G mmWave when all clients have an accuracy SLA of 0.02. The Static baseline is configured with fixed offloading interval 3 and batch size 3, while Clipper-like has a fixed offloading interval of 3, based on the aforementioned configuration search (§3.6.3). While all systems ensure that all clients meet their accuracy SLAs, ARISE supports on average 7.6 clients and significantly improves over other baselines by 2.6x–3.8x, validating that the performance gain of ARISE is similar as in the larger-scale emulation experiment. We note that the per-client average accuracy drops of Clipper-like and ARISE-α are only up to 0.011 and 0.013 respectively, which are far from the accuracy SLA of 0.02. This is due to the smaller batch sizes and shorter end-to-end latencies as the server in both baselines try to minimize extra delays caused by request conflicts, as we will discuss in §3.6.6.

System overhead. The proactive scheduler takes 0.8 ms per execution, which is negligible compared to the scheduler execution interval of every 200 ms. The depth estimation local tracker (warping) takes 3.8 ms on Pixel 5 and 4.0 ms on Pixel 2, while the accuracy estimator takes 2.0 ms on Pixel 5 and 2.7 ms on Pixel 2, both satisfying the real-time requirement at 60 FPS.

3.6.6 In-Depth Analysis

Benefits of proactive scheduling. The number of clients that a server can support depends on both client offloading frequencies and the server inference throughput, *i.e.,* the number of requests the server processes in a unit time, which in turn depends on both inference batch sizes and GPU idle times (periods when no DNN inference is performed). While reactive and proactive scheduling require clients to offload at similar frequencies, the server's performance is different due to the uncoordinated requests in reactive scheduling. We plot the average batch size and server idle time of ARISE vs. ARISE-α with accuracy SLA of 0.02 in Figure 3.17. While both scheduling algorithms select similar batch sizes for the clients, ARISE-α fails to reach the chosen batch sizes. In reactive scheduling, requests arrivals are uncoordinated and may arrive densely at times and sparsely at other times. When they arrive densely, the server will remove some clients to maintain the normal batch size that it can handle. When they arrive sparsely, the server cannot fill up the batch in time (and it cannot add clients quickly enough), and the resulting smaller batch size will lead to shorter end-to-end offloading latency and thus better task accuracy per-client. For the same reason, ARISE-α has a much higher percentage of GPU idle time (24.7%) compared to that of ARISE (8.8%), which is just below the 10% grace period (§4.5.5).

Server adaptation analysis. To see how ARISE adapts as the client content changes, in Figure 3.19 we plot the timeline of the average accuracy drop rate across clients, the average offloading frequencies across clients, the average batch size, and the number of concurrent clients with a 1 second moving window. Firstly, the average accuracy drop across clients shows high fluctuation over time, ranging from 0.007 to 0.013, which is due to changes in video content. Secondly, the average offloading frequency closely follows changes of the accuracy drop rate. The batch size is also affected by the accuracy drop rate. For example, the batch size reaches its peak value when the average accuracy drop rate drops to the lowest at around the 40-th second. By controlling both the offloading frequency and batch size accordingly as client accuracy drop rate changes, ARISE is able to dynamically adjust the number of supported clients (bottom figure) so that the number of supported clients is maximized while keeping accuracy drops within SLAs.

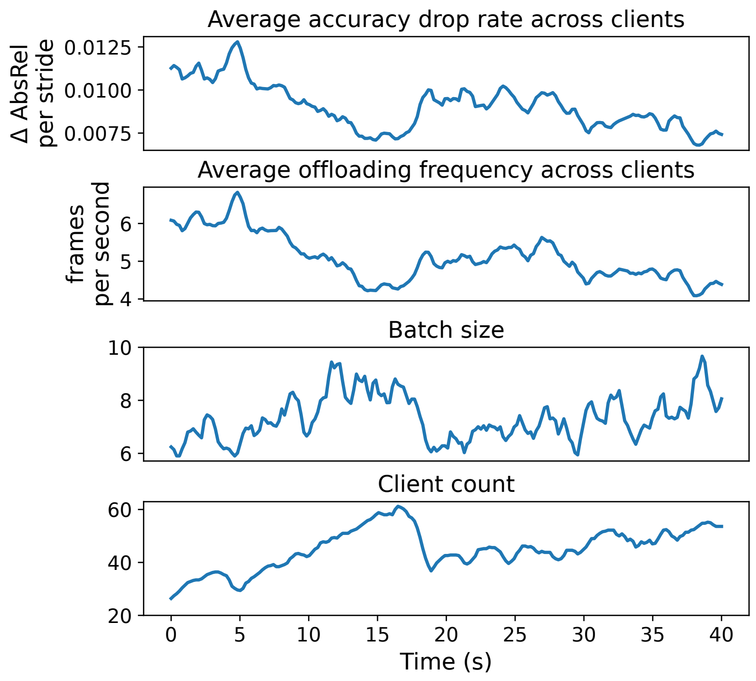


Figure 3.19. Timeline of server execution stats at accuracy SLA of 0.06.

Consistency between schedule and runtime. In practice, variations such as network dynamics may cause the runtime behavior to deviate from the generated schedule. We plot the difference between the scheduled start time and the actual start time of each batch in Figure 3.20. We see that most of the differences are below 0 (but greater than -16.7 ms), meaning that the batches start a little earlier than expected. This happens as the scheduled starting time assumes a buffering time (10% of inference time), but can start right away if all the expected requests for the batch have arrived. On the other hand, only 1.6% of the batches start later than the scheduled time, indicating that the runtime behavior closely follows the schedule by the proactive scheduler.

Accuracy estimation error. In Figure 3.21, we evaluate ARISE's accuracy estimator by measuring the difference between the estimated and actual average accuracy drops for each video and under different accuracy SLAs. The accuracy estimator shows minimal errors. For example, at accuracy SLA 0.06, the maximum estimation error is 0.008, which is below 15% of the accuracy SLA and is sufficient for guiding scheduler decisions. Furthermore, we notice

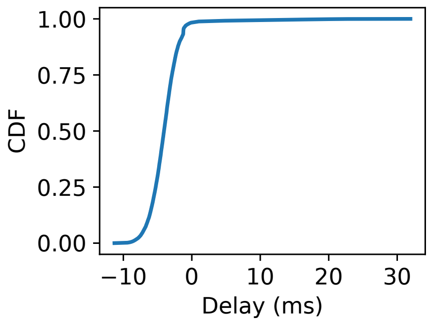


Figure 3.20. CDF of actual start time minus scheduled start time of the batches at accuracy SLA 0.06.

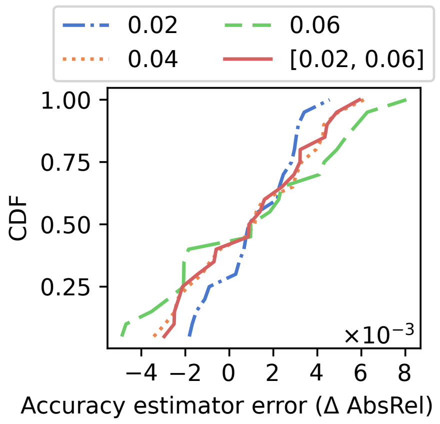
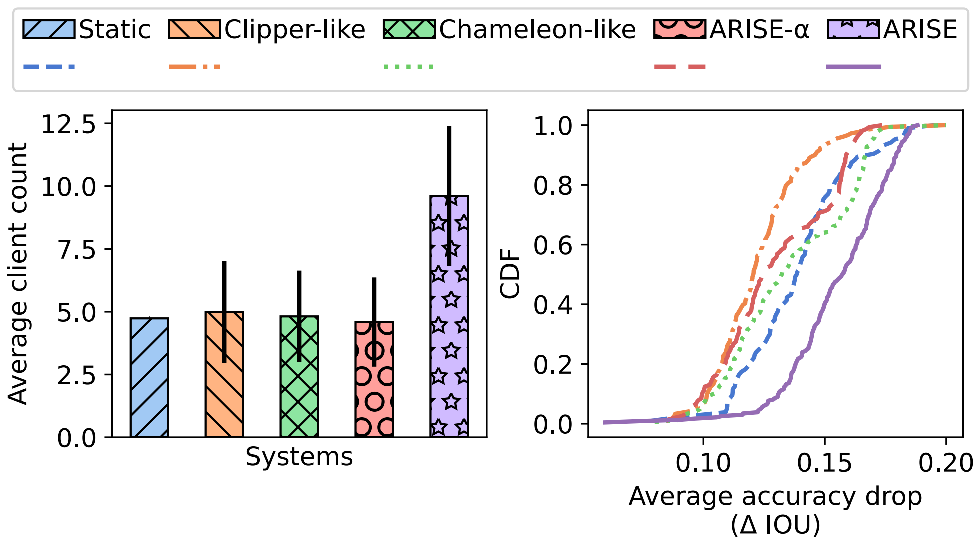


Figure 3.21. CDF of accuracy estimation error over different videos and under different accuracy SLAs. A positive value means over-estimation.

that the accuracy estimator tends to be more accurate under stricter SLAs. At accuracy SLA of 0.02, the estimation error narrows down to between -0.002 and 0.004. This is because clients offload more frequently under tighter SLA, which results in more recent frames being used for accuracy estimation (§3.4.3), and thus smaller estimation error.

3.6.7 Evaluation on Object Detection

To evaluate the generalizability of our framework, we evaluate ARISE against the baselines on a second task — object detection with emulated clients. Figure 3.22 shows the average number of supported clients and the CDF of per-client accuracy drops with all clients having an accuracy SLA of 0.2. The Static baseline is configured with fixed offloading interval 5 and batch size 5, while Clipper-like has a fixed offloading interval of 9. All baselines ensure that all clients meet the accuracy SLA. However, ARISE is able to support 9.6 clients on average, outperforming other baselines by 1.9x–2.1x, which shows the generalizability of ARISE to object detection. Compared to depth estimation, the object detection DNN model has a higher inference latency, and the benefit of batching diminishes faster (Figure 3.12).



(a) Average number of concurrent clients. (b) CDF of per-client average accuracy drops.

Figure 3.22. Performance comparison on the object detection task with accuracy SLA 0.2.

The characteristics of DNN models have an impact on ARISE's tradeoff between offloading frequency and batching. For example, in the object detection experiment in Figure 3.22, clients have an average offloading frequency of every 13.0 frames, and the average batch size is just 1.2. On the other hand, the depth estimation experiment in Figure 3.18 supports similar number of clients, but the average offloading frequency is every 4.6 frames and the average batch size is 3.1. ARISE prefers longer offloading interval over larger batch size in the presence of heavy DNN models, which exhibits its ability to dynamically adjust to different task characteristics. The difference in inference latency also explains the relatively higher number of clients supported by Chameleon-like, which does not support batching, compared to that in depth estimation.

3.7 Related Work

The large amount of prior work on DNN inference serving fall into generic inference serving, video analytics pipelines, or single-AR-client serving.

Generic DNN serving. As discussed in §3.3.1, the large number of generic DNN serving systems (*e.g.,* [28], [36], [93], [102]–[108]) serve requests from multiple clients but do not assume any correlation among requests. Additionally, cluster-level DNN serving optimizations, *e.g.,* auto scaling [93], assignment of requests to servers [105], and assignment of different tasks to servers [36], [105], are complementary to our work, which focuses on optimizing individual servers.

Single-client and multi-client video analytics. Similar to AR offloading serving, video analytics clients also offload a stream of frames and the frames are processed for video analytics tasks, *e.g.,* object detection, on the edge server. A major distinction between video analytics serving and AR offloading serving is that video analytics serving typically have relaxed latency requirements of hundreds of milliseconds, *i.e.,* analytics results on a frame do not need to be available in the current frame interval and are optimized for such latency-oblivious accuracy [30], [94], or are for retrospective analysis [97], [116]. Second, existing video analytics serving systems [14], [30], [94], [95], [98]–[101], [117] focus on dynamically adjusting *client-side* offloading configurations to optimize the accuracy under network dynamics or server resource constraints; they do not control *server-side* configurations, *i.e.,* batch size, by coordinating requests on the server. A few video analytic systems incorporate local tracking, but with a focus on a single client [38], [41]; these works can be augmented with ARISE to support multiple clients efficiently.

Single-client AR offloading serving. Many DNN offloading systems have been proposed for a single AR client for tasks such as object detection [9], [16], [31], human pose estimation [9], and depth estimation [10]. Such systems assume a dedicated server is used for each AR client.

Collaborative edge-assisted AR. A few collaborative edge-assisted AR systems optimize AR offloading serving by an edge server [118]–[121] by exploiting caching and serving cached results of previously offloaded frames. Such optimizations are applicable when AR clients encounter the same scene and are orthogonal to our design.

3.8 Summary

A cost-effective solution to deploying popular edge-assisted AR apps to support a large user base is to use MLaaS to serve offloaded AR inference requests. In this chapter, we presented to our knowledge the first framework that addresses the AR inference serving problem. The framework employs an online accuracy estimator that estimates the accuracy for each AR client under various configurations and an online scheduler that proactively coordinate requests from the clients served by a server. Our evaluation using a large set of emulated AR clients and a 10-phone testbed show that ARISE supports 1.7x–6.9x more clients compared to various baselines while keeping the per-client accuracy drops with the client-specified SLA.

4. IPIPE: EFFICIENT VIDEO ANALYTICS SERVING ON HETEROGENEOUS GPU CLUSTERS VIA POOL-BASED PIPELINE PARALLELISM

4.1 Introduction

Advances in machine learning in recent years for processing video streams [122], along with growth in Internet of things (IoT), edge computing and high-bandwidth access networks such as 5G have led to the wide adoption of video analytics systems [123]–[126] to support applications in diverse domain such as surveillance, transportation, public safety, healthcare, retail, and home automation.

In video analytics systems, streams of video frames from cameras deployed at different locations of interest are uploaded to the cloud servers that perform analytics, *i.e.,* deep neural network (DNN) model inference. Serving such inference requests is challenging for two primary reasons. First, video analytics inference requests often have stringent service level objectives (SLOs), *e.g.,* 200 ms [55], [95]. Second, inference requests from real-world applications can be bursty [127]–[129]. Meeting the latency SLO for all inference requests requires provisioning hardware resources for the peak load, which can be costly as the hardware resource becomes under-utilized during off-peak periods.

As with model training, model inference relies on the use of accelerators such as GPUs. With rapid innovation of GPUs [130], newer generations of GPUs have become available in the market in short release cycles. Yet, their high cost and limited supply have disincentivized cloud vendors and private organizations from retiring (older) low-class generations of GPUs. As a result, cloud vendors and private organizations are increasingly operating highly heterogeneous GPU clusters [24].

This chapter studies how to serve popular DNN models, *i.e.,* with high volumes of requests [131], on heterogeneous GPU clusters. Being able to do so not only allows utilizing low-class computing resources that are otherwise unusable [132] in clusters dedicated to model serving, *e.g.,* in edge clouds or private clouds running AI-based apps [133], [134], but also, as we will show in this chapter, can significantly enhance the serving throughput of high-class GPUs.

In particular, we explore the overlooked benefits of pipeline parallelism among low-class and higher-class GPUs in online model serving. While the benefits of pipeline parallelism have been well studied for throughput-oriented model training [132], [135], its potential for model serving under latency (SLO)-constrained settings has been largely unexploited. Intuitively, the benefits of partitioning a model and pipelining the partitioned inference among mixed high-class and low-class GPUs appear limited. For example, if the high-class GPU is 10× faster (*i.e.,* lower latency) than the low-class GPU for a given model, then simply running 1/10 of the model layers on the low-class GPU already leads to 1.9× longer total latency.

We instead make a key observation about the performance characteristics of DNN model inference on heterogeneous GPUs: *the two forms of diversity in model inference on a heterogeneous GPU cluster, diversity in model layers and in GPU types, can interact with each other synergistically.* First, a DNN model typically has many layers with diverse tensor shapes and sizes, which have varying GPU utilization on a given GPU [105], [136]. Second, more importantly, for the same DNN model, the relative per-layer inference latency on different classes of GPUs can vary significantly across the model layers. This observation suggests that partitioning the DNN model in a GPU-aware manner and executing each partition on the GPU type that runs most effectively can improve the effectiveness (throughput) of all GPU types and hence the inference throughput of the whole cluster. Effectively, higher-class GPUs are offloading part of the model inference to lower-class GPUs with minimum or no elongation of the end-to-end inference latency.

While the observation provides guidelines on efficient pipeline parallelism on heterogeneous GPU servers, partitioning a model and pipelining the inference of partitions along a *chain* of GPUs, as done in pipeline parallel DNN training [137], [138], is too stringent and will lead to suboptimal partitions: all partition stages need to have matching latency (to avoid pipeline stalls), which restricts the flexibility of model partitioning and leaves less opportunity for low-class GPUs to run layers they are efficient at.

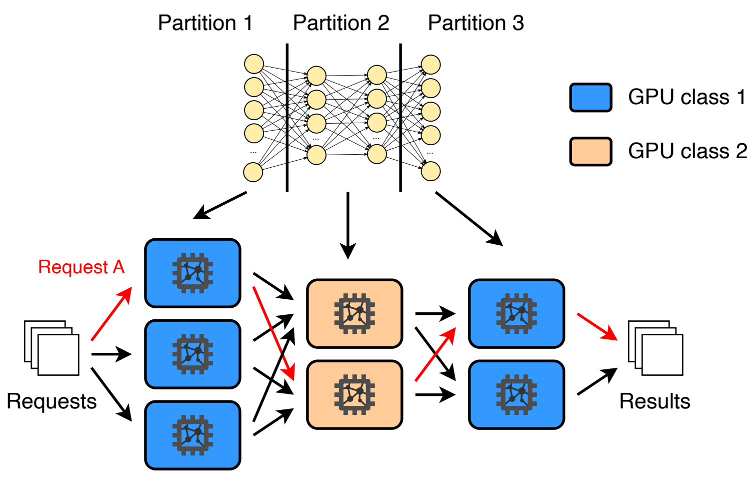


Figure 4.1. Pool-based pipeline parallelism on a heterogeneous GPU cluster with model partitioning. Each request (*e.g.,* request A) is processed by all partitions sequentially, and may be processed by any of the GPU servers allocated to each partition.

To this end, we present IPIPE, a model serving system that harnesses mixed GPU types in heterogeneous GPU clusters via pipelined model inference to maximize its serving throughput. IPIPE is built on three key ideas. First, to support maximal scheduling flexibility, it employs *pool-based pipeline parallelism* where each model partition is associated with a *pool* of GPU servers of the same class, and each request can be processed by any GPU allocated to each partition pool along the pipeline, as shown in Figure 4.1. This approach allows different partitions to have different numbers of GPU servers, with different inference latencies, and run with different batch sizes, as long as the inference throughput provided by each pool of GPU servers matches with each other.

Second, to realize the scheduling flexibility exposed by pool-based pipelined model inference, IPIPE generates the optimal configuration of pool-based pipelined model inference, *e.g.,* one that maximizes the model serving throughput of a given cluster while meeting inference SLOs, using Mixed Integer Linear Programming (MILP), which takes as input the per-layer inference latency on all candidate GPUs and under all candidate batch sizes from offline profiling.

There, however, exists a gap between such an MILP-based optimal solution, which in essence assumes ideal request arrivals, *i.e.,* synchronous arrival of batches in locksteps at all GPUs of the first partition pool, which flow down the pipeline partitions in sync, and the runtime. In practice, the inference requests arrive asynchronously and can be bursty [127]–[129], which results in batched requests being formed and injected into the cluster in succession, creating transient high load that overwhelms the throughput prescribed in the MILP solution, and introducing several sources of extra delay not accounted for in the MILP solution.

To bridge the gap between MILP solution and runtime dynamics due to asynchronous and bursty request arrivals, IPIPE treats the MILP-based formulation as the control plane which prescribes optimal DNN model partitions and GPU and network allocation, and employs a separate data plane that performs *resource reservation-based adaptive batching*, which overcomes a major limitation of adaptive batching used in previous work on pipelined inference serving. In particular, previous adaptive batching [28] decides on a batch size at each pipeline stage by optimistically assuming all the GPU and network resources downstream will be available, which can lead to infeasible batch sizes and miss the end-to-end inference SLO. In contrast, IPIPE uses a real-time resource scheduler to track and reserve GPU/network resources in the pipeline to ensure batches injected in the pipeline will meet their SLOs.

We evaluate IPIPE with Poisson-arrival workloads on top of both 100-GPU large-scale simulations and 16-GPU testbeds on Google Cloud consisting of a variety of high- and low-class GPUs such as NVIDIA V100, L4, T4, and P4. Evaluation across 18 CNN models shows that IPIPE achieves 44.4%–65.3% higher utilization of low-class GPUs while maintaining high utilization of high-class GPUs compared to various baselines, leading to 15.9%–64.0% higher serving capacity, while successfully processing 99% of the requests without dropping or SLO violations.

In summary, we make the following contributions:

- The first exploration of pipeline parallel model serving on heterogeneous GPU clusters under latency (SLO)-constrained settings.

- The complete design of IPIPE, which exploits pipeline parallelism to maximize inference throughput of heterogeneous GPU clusters. IPIPE employs three design ideas: pool-based pipeline parallelism, an MILP-based control plane that prescribes optimal pipeline plans, and a data plane that performs resource reservation-based adaptive batching to handle runtime dynamics due to asynchronous and bursty request arrivals.
- Extensive evaluation of IPIPE showing IPIPE outperforms baseline designs by 15.9%–64.0% in inference throughput and 44.4%–65.3% higher low-class GPU utilization.

We will open source IPIPE to facilitate further research on model serving on heterogeneous GPU clusters.

4.2 Motivation and Key Idea

We motivate how low-class GPUs can be effectively used to augment high-class GPUs in a heterogeneous cluster in model serving by exploiting pipeline parallelism.

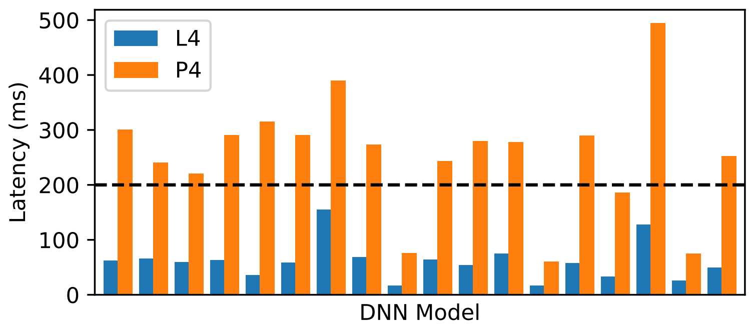


Figure 4.2. Inference latency of 18 popular DNN models (Table 4.2) under batch size 4 on different GPU classes.

Low-class GPUs fail to meet the inference latency SLO. On a highly heterogeneous GPU cluster, the inference time on old and low-class GPUs is usually several times longer than that on newer or high-class GPUs. For the 18 DNN models we use for evaluation (§4.7.1), the inference time (on the highly optimized TensorRT [139] inference framework) on the low-class NVIDIA P4 is 3.0x–7.9x longer than that on the high-class NVIDIA L4, as shown in Figure 4.2. Even if a model successfully runs on the low-class GPU without violating latency SLO, it can barely perform batched inference, which could significantly improve GPU utilization and throughput [28], [102], [103]. As shown in Figure 4.2, only

22% of the DNN models can run on the low-class GPU (NVIDIA P4) at batch size 4 without exceeding 200 ms, a latency SLO target commonly used among video analytics pipelines [55], [95].

Key insight: Diversity in per-layer inference delay across GPUs. To make use of low-class GPUs, given that running the entire model on the low-class GPU exceeds the latency SLO, the intuitive idea is when there is slack in running the whole model in the high-class GPU, simply partition the model and run a part of the model on a low-class GPU and the rest on high-class GPU while ensuring that the total inference latency remains below the latency SLO. The benefits of such an approach, if the model is partitioned in a GPU-oblivious manner, will be limited. For example, if the high-class GPU is 10× faster (*i.e.,* lower latency) than the low-class GPU for a given model, then simply running 1/10 of the model layers on the low-class GPU already leads to 1.9× longer total latency.

Our key observation is that there exist two forms of diversity in model inference on a heterogeneous GPU cluster: diversity in model layers and in GPU types, and they can interact with each other synergistically.

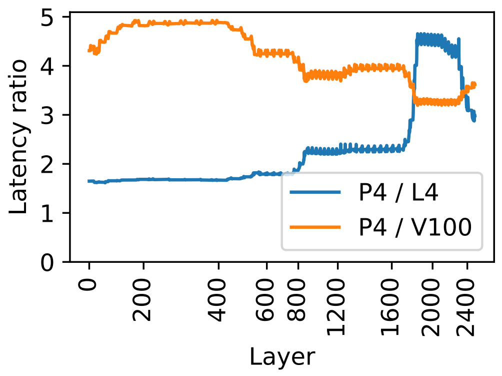


Figure 4.3. The ratio of inference latency on NVIDIA P4 over L4 and P4 over V100 across EfficientNet-B8 layers.

In particular, for many popular CNN backbone architectures, *e.g.,* EfficientNet [140] and ResNet [141], later layers have more channels compared to earlier layers, but with lower feature dimensions. Such architectural differences among layers within a DNN model can lead to different computational properties on GPU accelerators. To gain insight into this, we measure the ratio of the inference latency of the same layer on different GPU types for all the layers within a DNN model. Figure 4.3 shows that for EfficientNet-B8 on NVIDIA P4 over L4 and P4 over V100, respectively, with a moving window of 128 layers. The inference latency ratio on P4 over L4 is close to 1.0 for early layers, indicating these layers have similar inference latencies on both P4 and L4. On the other hand, later layers have much higher latency ratios, and those layers will suffer significant slowdown running on P4 over L4. If we were to partition the DNN model and run it on P4 and L4, we should place earlier layers on P4 and later layers on L4, which provides higher chances to keep the inference time below the latency SLO and enables batching opportunities. All 18 DNN models we studied (Table 4.2) exhibit varying latency ratios across layers and we omit the rest due to page limit.

Interestingly, the latency ratios between P4 and V100 show completely different trends on EfficientNet, where earlier layers exhibit much higher latency ratios than later layers. In this case, one would run on P4 later layers instead. Such differences in the trends of latency ratios happen due to GPU design tradeoffs, architectural improvements, and their interaction with DNN layers of different characteristics. For example, GPUs with more SMs or higher ops:bytes ratio provide more benefits for layers of larger size or higher arithmetic intensity [142]. As another example, Tensor Cores significantly improve the performance of layers involving matrix multiplications, but only for layers of certain shapes [143].

Such varying trends in per-layer latency ratios on different GPUs suggest that partitioning a DNN model in a *GPU-aware* manner is critical in exploiting pipeline parallelism so that high-class and low-class GPUs can work on model layers that they are optimized for, which improves their efficiency and hence the inference throughput of the whole cluster.

Compared to non-partitioned model inference, *e.g.,* on a single high-class GPU, partitioned model inference incurs the extra overhead of transferring feature maps at the partition points between GPUs. We observe that heterogeneous clusters such as Google Cloud and AWS come with high-bandwidth networks that theoretically can finish the transfer of a feature map in a fraction of the total inference latency. For example, P4 instances in GCP have a bandwidth of 32 Gbps, which can theoretically transfer feature maps of CenterNet (under batch size 1) which range from 3 MB to 50 MB under 0.8 ms–13.2 ms. However, it does imply that the DNN model should be divided into at most a few partitions as frequently switching between low-class and high-class GPUs during inference will be costly.

Key idea: pool-based pipeline parallelism. To apply model partitioning to exploit the diversity of per-layer inference ratio on different GPUs, a simple approach is partitioning a model and pipelining the inference of partitions along a *chain* of GPUs similarly as in pipeline parallelism DNN training [137], [138], and feature maps generated by one GPU are transferred to the next downstream GPU. Such isolated inference pipelines have the advantage of simple scheduling and coordination, but also come with two major drawbacks: (1) To avoid pipeline stalls, all partitions need to have similar inference latencies. However, such a partitioning strategy is too stringent and will lead to suboptimal partitions, *e.g.,* leaving layers with high latency ratios on the low-class GPU or having a large feature map at the partition point. (2) Many GPUs cannot enjoy the benefit of heterogeneous inference when the cluster contains more GPUs of one class than the other.

To provide more scheduling flexibility, we instead associate each partition with *a pool of GPU servers* of the same type, and each request can be processed by any GPU allocated to the first partition, and then continue the inference on any GPU in the second partition, and so on, as shown in Figure 4.1. This approach mitigates the drawbacks of isolated pipelines: different partitions can have different numbers of GPU servers, have different inference latencies, or even run with different batch sizes, as long as the inference throughput provided by each pool of GPU servers matches with each other, and the total latency is below the latency SLO. In an optimal partitioning, multiple such pool-level pipelines may be employed at the same time, employing different ways of partitioning the DNN model running on different GPU pools.

4.3 Prelude to IPIPE: Basic MILP Formulation

The primary challenge in exploiting pipeline parallelism in pool-based pipelined inference is to figure out the optimal way to partition a DNN model, the placement of the DNN partitions onto GPU servers, and the batch size for the GPUs in each partition. It is relatively straightforward to formulate an MILP problem to figure out the optimal solution. We briefly describe the MILP formulation below and leave the full mathematical formulation in §4.A.

Inputs. The MILP formulation takes as input the GPU count of each GPU type, the interconnect bandwidth of the target cluster, and the inference latency SLO. To decide the optimal model partitions, it also requires the inference latencies of individual layers within a DNN model under different batch sizes and on different GPU models, as well as the intermediate feature map sizes, both of which can be obtained through the profiling output of TensorRT [139].

Encoding model partition and placement. Suppose the GPU cluster consists of 2 GPU types and we restrict a DNN model to be divided into at most 3 partitions. The placement of DNN model partitions falls into one of 14 potential *pooled pipelines*: if partitioned into 2 partitions, each partition can run on a pool of either GPU types (4 pipelines); if partitioned into 3, each of the 3 partitions again has the choice to run on a pool of either GPU types (8 pipelines); the DNN model can also directly run on a pool of either GPU type without partitioning (2 pipelines). In an optimal solution, multiple pipelines may be employed at the same time, *e.g.,* when a DNN model should be partitioned onto 2 GPU types and there is an excessive number of one GPU type, the extra GPUs may need to run the DNN model without partitioning.

For each partition within a pipeline, we need to decide on the exact partition points, *i.e.,* the first and last layers. To this end, we construct a set of binary decision variables for each partition indicating whether a layer is the first or last layer of a partition. Apart from that, we also create decision variables to represent the batch size and the number of GPUs used by each partition.

Constraints. The total latency of each pipeline, including the inference latency of each partition and the feature map transfer latency between partitions (both can be derived from the batch size), should be below the latency SLO; the total GPU count used by all partitions pertaining to a specific GPU type should not exceed the GPU count available for that GPU type. Finally, the inference throughput of a pipeline is bottlenecked by the partition of lowest throughput, where each partition's throughput can be calculated based on its batch size, inference latency, and the number of GPUs allocated to the partition.

Objective. By default, we try to maximize the total inference throughput of the GPU cluster, which is the sum of the throughputs of all pipelines employed by the MILP solution. The MILP formulation can also be configured for other objectives like minimum server cost [96] or provisioned power [105]. In the presence of multiple DNN models, given the ratio between the DNNs' workloads, the MILP formulation computes the normalized throughput for each DNN (throughput divided by the DNN's workload percentage), and maximizes the lowest normalized throughput among the DNNs.

Outputs. After solving the optimization problem, the MILP formulation outputs the pipelines employed by the optimal plan, *i.e.,* those being allocated at least 1 GPU. For each pipeline, the solver outputs the DNN model partition points, the batch size used by the GPUs in each partition, and the number of GPUs allocated to each partition.

Intuitively, the pipelines and partition points selected by the MILP solution are affected by the cross-GPU latency ratios of the DNN layers (§4.2), as well as the intermediate feature map sizes, as larger feature map size takes longer to transfer and leaves less time for inference (§4.5.3). Finally, the partition points, batch sizes, and GPU counts for each partition are chosen collectively to ensure partitions within the same pipeline have matching throughputs.

4.4 Challenges in Developing a Working System

While the MILP formulation above provides the optimal plan in theory, turning it into a working DNN serving system faces several practical challenges, as discussed below.

C1: Extensive search space of the MILP formulation. The MILP formulation needs to decide the first and last layers of a DNN partition, whose complexity depends on the number of layers in a DNN model. For the set of representative models in our evaluation (§4.7.1), the average layer count is 613.2. The partition points need to be searched for all partitions across all pipelines, making the search space combinatorial. The search space is further inflated by additional dimensions including inference batch size and GPU count used by each partition. With such a vast search space, it takes more than 7 hours (running the Gurobi [144] solver on a Google Cloud n1-standard-64 instance) to obtain the optimal solution for 80 layers, making it impractical to adapt to changing workload, *e.g.,* diurnal load [105].

C2: Asynchronous and bursty request arrival. In essence, the MILP formulation outputs a solution that assumes ideal inference request arrival. Suppose a pipeline solution consists of two partitions with 40 ms inference latency each, and the inference throughput is 1000 requests per second. The MILP solution effectively assumes that 40 requests arrive at the same time every 40 ms, which are simultaneously processed by all GPUs allocated to the first partition, and then forwarded to the second partition, and so on.

In reality, in an online inference system, inference requests arrive asynchronously and in a bursty manner, which can disrupt the MILP solution with two forms of extra delays: (1) Early arriving requests have to wait for later requests to form a batch to be dispatched to a GPU in the first partition, incurring *initial batching delay* (D1); (2) The staggered batched inference at GPUs in the first partition will cascade down the partitions in the pipeline. In such staggered pipelined inference, it is possible when a GPU in partition i finishes inference on a batch, all of the GPUs in partition $\mathrm{i}+1$ are still busy running other batches, causing *inter-partition queuing delay* (D2). Such queuing delay is further complicated when partitions use different batch sizes, requiring the split and merge of batches which creates complex dependencies between the GPUs of different partitions.

To incorporate the above extra delays at runtime, we could add a predefined margin to the latency SLO as input to the MILP formulation [28], [106] which will output adjusted (still fixed) batch sizes. But simply adding a static margin cannot handle bursty request arrival, which can result in either too many or too few transient requests compared to the adjusted target batch size. Such dynamic conditions require dynamically adjusting the batch sizes.

C3: Network contention. In practice, it is common that multiple GPUs collocate on the same server and share the network bandwidth of the server. When they send or receive intermediate feature maps at the same time, the network contention leads to extra transfer delay (D3). The problem becomes more severe when we divide each GPU into multiple virtual GPUs (§4.5.4), which increases the number of “GPUs” on the same server that likely transfer feature maps at the same time. Note that this issue cannot be addressed by conservatively allocating each virtual GPU an equal share of the available bandwidth, as it results in low bandwidth for all virtual GPUs, significantly limiting the benefit of pipeline parallelism.

4.5 IPIPE Design

4.5.1 Design Rationale

As discussed above, designing a practical pipelined inference serving system on a heterogeneous GPU cluster faces a key challenge: the MILP formulation does not capture or handle runtime dynamics due to asynchronous and bursty request arrival or delayed feature map transfer from network contention. We tackle these challenges by splitting IPIPE, our pool-based pipelined model inference system for heterogeneous GPU clusters, into a control plane and a data plane. First, IPIPE treats the MILP-based formulation as the control plane that prescribes optimal DNN model partitions and GPU allocation. Second, to handle delays caused by asynchronous and bursty request arrivals (D1 & D2) and network contention (D3), IPIPE employs a novel data plane that performs resource reservation-based adaptive batching to ensure the request batches injected into the pipeline meet their latency SLOs.

4.5.2 Architecture Overview

Figure 4.4 shows the architecture of IPIPE, which consists of an offline phase, a control plane, and a data plane.

Offline phase. In the offline phase, the DNN model profiler profiles a model's per-layer inference latency on different GPU types. To reduce the search space of the MILP solver (C1), we design a pre-partitioning method that groups the layers of each DNN model into blocks which are fed into the MILP solver (§4.5.3). With this method, the MILP solver only needs to find partition points among a few blocks instead of hundreds of layers, significantly reducing the search space.

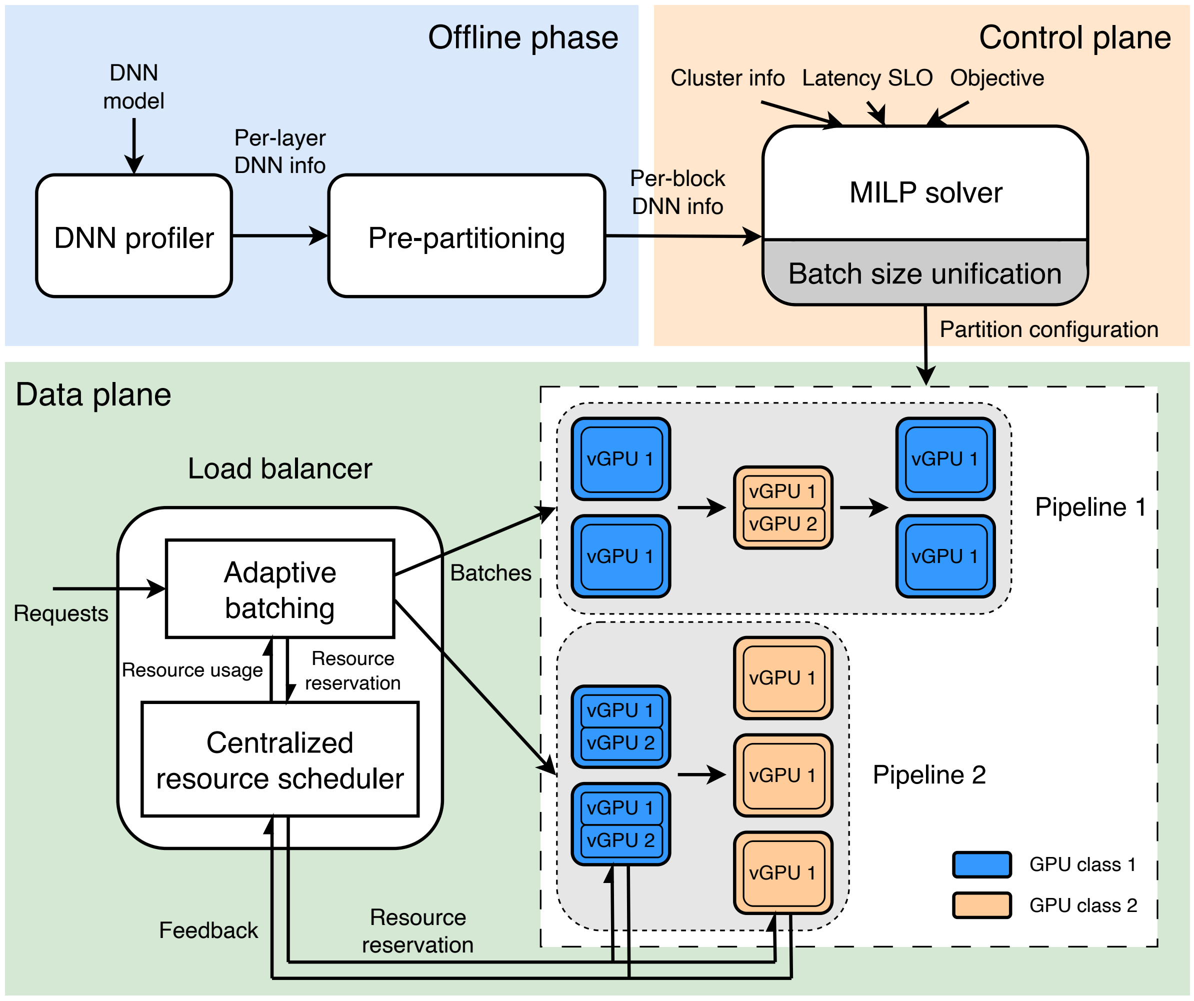


Figure 4.4. IPIPE architecture.

Control plane. The control plane, which runs the MILP solver, takes as input the profiling information of DNN blocks and the inference latency SLO for each DNN, the cluster information (GPU count for each GPU model), along with high-level objective, *e.g.,* maximum throughput, and outputs the partitioning of DNN models and allocation of GPU resources across partitions and pipelines (detailed in §4.3). The MILP solver runs periodically, triggered dynamically on demand by changes in workload, *e.g.,* when load ratio changes in serving multiple DNNs, or in server resources, *e.g.,* when the cluster in a public cloud is scaled up and down. We observe that the synchronization among partitions (**C2**) could be much simpler if partitions within the same pipeline all use the same batch size. As such, we enhance the MILP formulation with *batch size unification*, which is detailed in §4.5.4.

Data plane. The data plane groups inference requests into batches and executes them through the pools of GPUs in each of the pipelines prescribed by MILP. To address the delays discussed in C2 and C3, we perform adaptive batching that dynamically adjusts the batch size based on the amount of time left for inference. A simple adaptive batching scheme for pool-based pipeline parallelism inference is to place an adaptive batching scheduler before each pool of GPUs, which receives requests from the last stage of the pipeline and adaptively forms batches for the current stage, based on the remaining time until the SLO deadlines. However, such a local, *reactive* design assumes both computing and network resources for the remaining stages are/will be available, which may not be true due to bursty request arrivals and network contentions, leading to SLO violations. Furthermore, the scheduler often cannot detect SLO violations until later stages of the pipeline, which not only results in high request drop rate, but also wastes computing resources.

To overcome these drawbacks, we design a novel, resource reservation-based adaptive batching scheme, which has two synergistic components: a resource scheduler, and an adaptive batching component. The resource scheduler manages the available GPU and network resources in the pipeline, based on which the adaptive batching component selects the best path in the pool-based pipeline, denoted as a *pipeline path*, and the batch size. The resources on this pipeline path are in turn reserved by the resource scheduler. This resource reservation scheme addresses D1 (initial batching delay) and D2 (inter-partition queuing delay), as the amount of time spent between partitions becomes predictable, allowing the adaptive batching algorithm to select the batch size to avoid end-to-end latency SLO violations. Additionally, D3 (network contention delay) is mitigated as reserving the network resources prevents concurrent network transfers to/from the same host.

4.5.3 DNN Pre-Partitioning

As discussed in **C1**, the sheer amount of layers within a DNN model results in a huge search space for the MILP formulation and prevents the solver from finding good plans in a reasonable amount of time. To this end, we devised a simple DNN pre-partitioning approach that groups the layers in a DNN model into a few (N) blocks of approximately equal runtime. Specifically, we start from the first layer and sequentially group consecutively layers together until their combined runtime is as close as possible to $1/N$ of the runtime of the entire DNN; this process is repeated until we reach the last layer. After grouping layers into blocks, we profile the blocks on different GPU types and with different batch sizes, as needed by the MILP solver. It takes only less than 10 minutes to profile a single DNN model, as each block can be profiled independently; when solving MILP, we obtain the latency of each partition by adding up the latencies of the blocks in it.

With pre-partitioning, the MILP solver only needs to find partition points among the N blocks (we tried $N = 5$ to $N = 20$ and found $N = 10$ provides a good balance between plan optimality and MILP running time), instead of 613.2 layers on average across the set of models (§4.7). As a result, the MILP runtime is significantly reduced to 4.6 seconds on average over different DNN model and GPU cluster setups, enabling the control plane to adapt to changing workloads.

Alternative approach. We also devised a more systematic approach to pre-partitioning by formulating it into another MILP problem, which minimizes the total feature map size at all block boundaries while ensuring that the blocks have similar inference latencies. However, our evaluation shows this systematic approach has minimal gains (about 1% in terms of the total throughput) over the simple equal partition approach described above. The reason is that the equal partition approach only leads to significantly larger feature map sizes at some of the block boundaries, while the control-plane MILP solver can often generate a partition plan that avoids partitioning at those suboptimal block boundaries. Therefore, we opted for the equal partition approach for simplicity.

4.5.4 Batch Size Unification

We tackle the key challenge of the data plane — bridging the gap between MILP solution and runtime dynamics (**C2** and **C3**) — in two steps. In the first step, we simplify the challenges faced by data plane scheduling with batch size unification.

As discussed in **C2**, mismatch of batch sizes between partitions within a pipeline requires batches to be merged and split which complicates scheduling of batched inference across partitions. Things can be substantially simplified if all partitions within the same pipeline use the same batch size. However, as the GPUs have different computational capacities, in the plans generated by the MILP planner, high-class GPUs tend to use larger batch sizes compared to low-class GPUs to improve GPU utilization and inference throughput. Naively forcing all partitions within the same pipeline to use the same batch size can significantly impact system performance.

We approach this dilemma from a different perspective: instead of forcing all partitions of a pipeline to use the same batch size, what if we vary the GPU size to equalize the batches per GPU? One way to do this is by adjusting the pipeline plan output by MILP. Suppose the MILP-generated plan says partition 1 uses batch size 8 and partition 2 uses batch size 16, we could adjust the plan by dividing each GPU for partition 2 into two and running inference on each "virtual GPU" with batch size 8. This approach is relatively straightforward but has two drawbacks. First, when the batch size is not a multiple of the other, the MILP generated plan has to be adjusted to use smaller batch sizes to satisfy the multiple relationships between batch sizes which not only becomes suboptimal due to underutilizing the GPU, but also may lead to mismatch between per-partition throughputs within the pipeline. Second, our measurement shows that dividing up a GPU into virtual GPUs also produces slightly different (both shorter and longer) inference latencies compared to running the equivalent batch size without dividing the GPU, which further complicates plan adjustment.

Instead of making post-adjustments to the MILP plan, we directly incorporate virtual GPUs and the batch size constraints into the MILP formulation, so that the MILP solver can take into account the throughput and inference latency differences between batches on different partitions and make holistic decisions. To this end, instead of feeding a GPU as a whole to MLP, we feed four possible virtual GPU types: 1/1, 1/2, 1/3 and 1/4 of a physical GPU (this is achieved with Multi-Process Service (MPS) [145] during runtime). The use of virtual GPUs only mildly expands the search space of the MILP solver as there are only 4 virtual GPU types. Besides, we profile the per-block inference latencies under not only different batch sizes and GPU types, but also different virtual GPU types.[1] Finally, we add additional constraints to the MILP formulation requiring all partitions within the same pipeline to use the same batch size, which omits the need to merge/split batches during pipelined inference at runtime. The detailed mathematical representation of the enhanced MILP formulation is provided in §4.B.

4.5.5 Resource Reservation-Based Adaptive Batching

While in §4.5.4 we unify the partitions within the same pipeline to use the same batch size, serving asynchronous and burst inference requests at runtime still requires dynamically forming and scheduling batched inferences across the pipelines, *i.e.,* dynamic batching. As discussed in §4.5.2, a simple reactive approach to adaptive batching for pipelined inference assumes both computing and network resources for the remaining stages are/will be available, which may lead to SLO violations and wasted computing resources.

To avoid the above drawbacks, we design a novel, resource reservation-assisted adaptive batching scheme. The scheduler maintains a global view of the resource availability and keeps records of the time slots when each resource is available. Next, the scheduler interacts with the adaptive batching component to help it determine the optimal batch size and the best path in the pipeline based on resource availability. Finally, the batch is dispatched to the pipeline and the resources on this path are in turn reserved by the resource scheduler.

Resource reservation. The resource scheduling algorithm works at the batch level. Thus, it needs to be fast to keep up with the request arrival rate. To this end, we design a greedy algorithm that probes and decides in real-time the computing and network resource to use for a new batch in a candidate schedule via two functions as shown in Algorithm 3. Given a pipeline and a hypothetical batch, `probe()` determines the optimal reservation for that batch, *i.e.,* which virtual GPUs and their network links will be used for that batch, *i.e.,* the pipeline path, and when each resource on the path will be used, with the goal of finishing the batched inference as early as possible; `reserve(r)` applies the given *reservation* r, *i.e.,* updating the reservation tables kept by the scheduler. Since `probe()` is based on real-time

[1]↑We capture the interference between virtual GPUs during profiling by running the same DNN on all virtual GPUs of the same physical GPU in parallel.

Algorithm 3: Resource reservation functions.

```
function probe(pipeline, bs)
    $t_g$ = now(), $r_g$ = [];
    for partition in pipeline do
        $l_n$ = calcNetLat(partition, bs);
        $l_\mathrm{i}$ = calcInferenceLat(partition, bs);
        $t^* = \infty$, $r^*$ = [];
        for gpu in partition do
            $t = t_g$, $r$ = [];
            // est. time to transfer feature map
            if not first partition then
                $u$ = lastGpu.netUL, $d$ = gpu.netDL;
                $t$ = earliestSlot([$u$, $d$], $t$, $l_n$);
                $r$ += [{$u$, $t$, $l_n$}, {$d$, $t$, $l_n$}], $t$ += $l_n$;
            // est. time to finish inference
            $t$ = earliestSlot([gpu], $t$, $l_\mathrm{i}$);
            $r$ += [{gpu, $t$, $l_\mathrm{i}$}], $t$ += $l_\mathrm{i}$;
            if $t < t^*$ then
                $t^* = t$, $r^* = r$, gpu$^*$ = gpu;
        $t_g = t^*$, $r_g$ += $r^*$, lastGpu = gpu$^*$;
    return $r_g$; // resource usage

function reserve($r_g$)
    for {res, start, dur} in $r_g$ do
        markReserved(res, start, dur);
```

resource availability, it directly takes into account extra delays D2 (inter-partition queuing delay) and prevents D3 (network contention delay). The pipeline and batch size arguments are chosen by the adaptive batching algorithm described later, which takes into account D1 (initial batching delay).

`probe()` works as follows. For each partition within the pipeline, we go through all virtual GPUs allocated to the partition and determine placing the batch on which virtual GPU gives the earliest completion time for that partition stage (t^*) and reserve the resource for the time and duration (r^*). Since each partition stage other than the first partition consists of two steps — receiving the intermediate feature map from the previous partition and performing inference, for each candidate virtual GPU, we add up the earliest completion time for downloading the feature map, after which the earliest completion time for the inference. We implement this using a helper function `earliestSlot(res, t, l)`, which returns the earliest time (no earlier than t) when a list of resources res are free for duration l. Since feature map transfer requires the network resources on both sending and receiving sides to be available at the same time, we use `earliestSlot(res, t, l)` to find the earlier available transfer slot that works for both the last GPU's uplink and current GPU's downlink (lines 10–12). After reserving the two links for feature map transfer, we update current time t and find and reserve the earliest available inference time slot for the current GPU (lines13–14). The resource allocation algorithm runs in linear time in the number of virtual GPUs allocated to the pipeline.

Resource reservation-assisted adaptive batching. Adaptive batching is widely employed by prior work to avoid latency SLO violations during runtime, by dynamically adjusting the batch size based on the closeness of the requests' deadlines [28], [103]. We face two challenges in applying adaptive batching in our system: (1) The optimal batch size depends on which pipeline the batch is sent to; (2) The optimal batch size for a chosen pipeline depends on resource availability. We overcome both challenges by utilizing our resource reservation functions described above. Algorithm 4 shows our resource reservation-aware adaptive batching algorithm.

We start by deciding on the pipeline by probing which pipeline can finish a batch of its unified batch size with the least amount of *waiting time* given the current resource availability, where the waiting time is calculated as the sum of time spent waiting for each needed resource in the pipeline path returned by `probe()`. While this approach does not necessarily select the optimal pipeline, as it does not consider the properties, *e.g.,* the deadlines of pending requests, our experiments show that requests are distributed to different pipeline in proportion to their expected throughputs in the MILP solution.

Next, we decide the pipeline path and the batch size for the selected pipeline by probing the completion time for batches of different batch sizes. In particular, we keep decreasing

Algorithm 4: Resource reservation-aware adaptive batching.

```
1  while true do
2      $q$ = pending requests sorted by arrival;
       // choose the pipeline
3      $t^* = \infty$;
4      for pipeline in pipelines do
5          $r$ = probe(pipeline, pipeline.bs);
6          if waitTime($r$) $< t^*$ then
7              $t^*$ = waitTime($r$), pipeline$^*$ = pipeline;
       // choose the batch size
8      for bs = pipeline$^*$.bs down to 0 do
9          $r$ = probe(pipeline$^*$, bs);
10         if finishTime($r$) $\leq q[0]$.deadline then
11             break;
       // choose the action
12     if bs == 0 then
13         drop $q[0]$;
14     else if $q$.length < bs then
15         wait for more requests till requests in $q$ are about to miss deadline;
16     else
17         reserve($r$);
18         dispatch first bs requests in $q$ according to $r$;
```

the batch size starting from the batch size in the MILP solution and hence the inference latency, until the completion time returned by `probe()` is before the deadline of the first (oldest) pending request. Finally, one of three actions is taken depending on the chosen batch size and the number of pending requests. If the deadline cannot be met even with batch size 1, the oldest request will be dropped and the adaptive batching process starts over; if the number of pending requests is smaller than the chosen batch size, the scheduler waits for more requests (till the last moment when the requests in queue can still be processed without SLO violation if no more requests are coming in); otherwise, the resources returned by `probe()` are reserved and the requests at the head of queue are grouped into a batch of the chosen batch size and dispatched to the virtual GPUs according to the resource reservation.

Since the batch size is based upon the actual remaining time of the requests in the queue, the extra delay D1 (initial batching delay) is taken into account.

The pseudocode for our adaptive batching algorithm is provided in **??**.

Feedback correction. The resource reservation functions maintain resource reservation tables that keep track of when each resource will be used. However, this scheduler's view of resource usage might deviate from reality due to variations in inference time and network bandwidth. To this end, we let all workers report back to the scheduler when the reserved resources were actually used immediately after every resource usage. The scheduler updates the resource usage table accordingly. The feedback correction mechanism ensures that the scheduler's view of resource usage is synchronized with reality at all times.

Extra SLO margin in the control plane. While our resource reservation-based adaptive batching algorithm ensures requests meet their SLOs, we notice that the resulting batch size may be much smaller than that in the MILP output due to extra delays D1–D3, causing large deviations from the MILP plan. To bridge the gap between control plane planning and data plane execution, we deduct an empirically determined margin from the SLO when running the control plane MILP solver, so that the adaptive scheduler picks batch sizes the same as in the MILP output most of the time.

4.6 Implementation

Offline phase and control plane. We implement the offline phase and control plane in Python in 2.7 kLOC. We use Gurobi [144] as the MILP solver. We work with DNN models in their ONNX format and TensorRT format interchangeably, since the ONNX format provides flexibility, while the TensorRT format provides high inference performance.

Data plane. We implement both a discrete-event simulator capable of simulating large-scale GPU clusters, as well as a real-world implementation for IPIPE's data plane, in about 9.0 kLOC. The simulator is implemented in Java, and the real-world implementation is in a combination of Julia and C++. For the real-world implementation, the inter-node communication is implemented with TCP for control messages and NVIDIA NCCL for feature maps. To minimize the feature map transfer latency, we quantize float32 feature maps to float16 (only at partition boundaries), effectively reducing the transfer size by half. We find such quantization has negligible impact on the inference accuracy. For example, the accuracy dropped by 0.00%, 0.01%, 0.01% for object recognition, object detection, and instance segmentation tasks, respectively.

4.7 Evaluation

In this section, we evaluate IPIPE's serving performance under a variety of DNN models from different tasks, considering various combinations of low-class and high-class GPUs. We show that IPIPE can serve 15.9%–64.0% more requests compared to various baselines while meeting 99% SLO attainment, on the discrete-event simulator with 100 GPUs. Additionally, on 16-GPU clusters deployed on Google Cloud, IPIPE achieves 16.7%–52.8% higher serving throughput. We also conduct sensitivity analysis to show the impact of GPU composition and SLO on IPIPE's performance.

4.7.1 Methodology

Table 4.1. Heterogeneous Cluster (HC) setups.

Setup	GPUs	Setup	Instances	GPUs	BW (Gbps)
HC1-L	25× L4, 75× P4	HC1-S	4× `g2-standard-16`, 2× `n1-highcpu-16`	4× L4, 12× P4	50
HC2-L	25× L4, 75× T4	HC2-S	1× `g2-standard-48`, 6× `n1-highcpu-32`	4× L4, 12× T4	32
HC3-L	25× V100, 75× P4	HC3-S	2× `n1-highcpu-16`, 12× `n1-highcpu-16`	4× V100, 12× P4	50
HC4-L	25× V100, 75× T4	HC4-S	1× `n1-standard-64`, 6× `n1-highcpu-32`	4× V100, 12× T4	32

Cluster configuration. We consider 4 heterogeneous cluster setups, labelled as HC1−HC4 in Table 4.1. Each setup consists of a large (L) 100-GPU variant used for the discrete-event simulator, and a small (S) 16-GPU variant deployed on Google Cloud. Note that a Google Cloud VM instance can host multiple GPUs, resulting in each HC having a varying number of VMs while maintaining a consistent number of GPUs. Accounting for the scarcity of high-class GPUs [24], our default configuration includes 25 high- and 75 low-class GPUs for each HC's large variant, and 4 and 12 for the small variant. We further evaluate IPIPE's performance under different ratios of high- and low-class GPUs in §4.7.6. Note that for GPU-equipped VMs, Google Cloud provisions network bandwidth based on the number and type of its GPUs, leading to different interconnect bandwidths across HCs. Furthermore, the effective bandwidth for both large and small clusters is only 1/5 the claimed values in Table 4.1 to accommodate the observed 5× network tail latency on Google Cloud.

Table 4.2. DNN models used in the evaluation.

Recognition	Detection	Segmentation	Others
ConvNext [146]	ATSS [147]	APCNet [148]	Color-v2 [149]
EfficientNet [140]	CenterNet [150]	DNL-Net [151]	
GoogleNet [152]	FSAF [153]	EncNet [154]	
RepVGG [155]	GFL [156]	FCN [157]	
WideResNet [158]	RTMDet [159]	GCNet [160]	
	EfficientDet [161]	NonLocalNet [162]	

DNN models. We select 18 DNN models from public DNN registries such as TorchVision [163], OpenMMLab [164], and OpenVINO model zoo [165]. The selected DNNs serve a variety of popular computer vision tasks, as shown in Table 4.2.

Metrics. We employ two key metrics to evaluate the inference serving capability. First, *SLO attainment* represents the percentage of requests that are successfully processed without being dropped or violating the SLO, under a specific offered load. Second, we measure the maximum load that the system can handle at 99% SLO attainment.

Baselines. We compare IPIPE against various baselines as described below. For fair comparison, we use IPIPE's data plane, *i.e.,* the resource reservation-based adaptive batching (§4.5.5), with all baselines.

- **No-Partitioning (NP)**. NP executes the entire DNN on either high-class or low-class GPUs without partitioning. The allocation is done by solving IPIPE's MILP formulation without model partitioning. When integrated with IPIPE's centralized scheduler, NP effectively performs adaptive batching to dispatch the largest possible batch to the next available GPU while meeting the SLOs for each request in that batch. This way of serving DNNs on a heterogeneous cluster is representative of various prior works [93], [96], [105], [107], [127], [166]–[168].

- **DART-r**. DART [136] is a DNN inference framework that can partition a DNN onto heterogeneous CPU and GPU cores. However, vanilla DART assumes each partition is served by a single GPU, limiting the number of partitions to the number of devices, and its usage to scenarios with only a few GPUs. To address this limitation, we introduce DART-r, a modified version that **r**eplicates DART configurations applied to pairs of low- and high-class GPUs (more efficient than longer pipelines from fewer feature map transfers). In cases where one class has more GPUs than the other, the leftover GPUs are allocated to run entire DNNs without partitioning. Finally, quantization is applied to the partition boundaries in the same way as in IPIPE.

Setup. Following prior work [129], for each DNN, we set the default SLO to be 5× its inference latency on the fastest GPU (NVIDIA L4) at batch size 1; we further evaluate IPIPE under other SLOs in §4.7.6. As mentioned in §4.5.5, to bridge the gap between control plane planning and data plane execution caused by the extra delays, a 40% margin is deducted from SLO as input to DART-r and IPIPE's MILP formulation, as well as in NP (which does not have D2, but effectively longer D1) for picking the maximum batch sizes that satisfy the SLO. In the comparison of IPIPE with the baselines, we use load factor 1.0 to denote the serving throughput in the output of IPIPE's MILP. We generate Poisson-arrival requests with an average request rate (λ) ranging from 0.05 to 1.0 times the load factor, at an interval of 0.05. For each λ, the experiment lasts 30 seconds.

4.7.2 End-to-end Results

Overall results. In this section, we evaluate IPIPE's capability of serving DNNs over 100-GPU clusters (HC1-L to HC4-L) on the discrete-event simulator. We randomly divide the 18 DNNs into 6 groups of 3 DNNs each (G1–G6), and serve DNNs within each group in parallel. DNNs within each group are assumed equal amount of incoming workloads.

During runtime, we record the maximum load ratio that each DNN can achieve under 99% attainment.

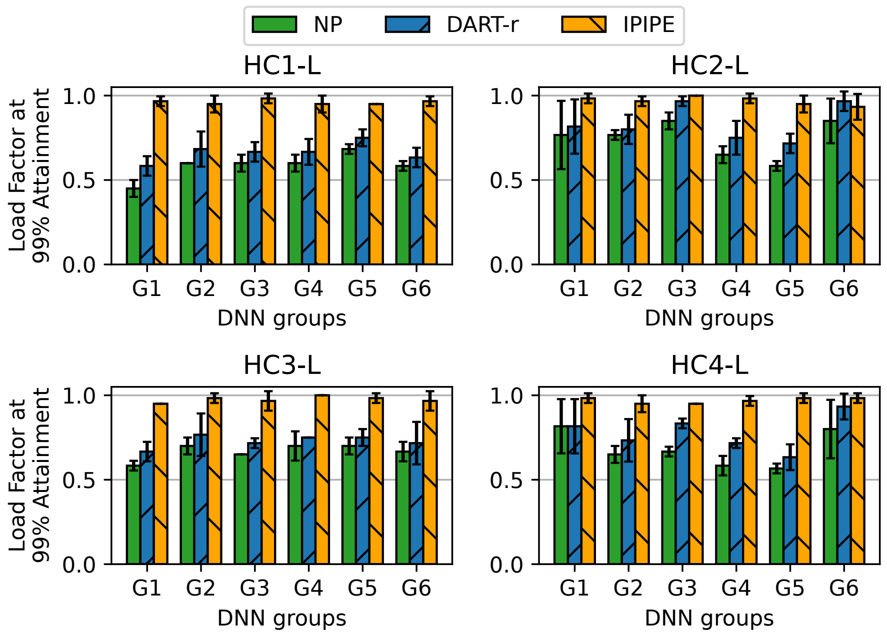


Figure 4.5. The maximum load factor each system can achieve under 99% SLO attainment on 100-GPU clusters.

Figure 4.5 shows under cluster configurations HC1-L to HC4-L respectively, the maximum load factor achieved under 99% SLO attainment, averaged over the DNNs in each group. Firstly, we find that IPIPE consistently outperforms NP. Across HC1-L–HC4-L, IPIPE achieves 64.0%, 30.2%, 46.2%, 42.4% higher load factors than NP, showing the advantage of IPIPE's model-parallelism inference. Secondly, compared to DART-r, IPIPE achieves a 44.8%, 15.9%, 34.0%, and 24.6% higher load factors, showing the advantage of IPIPE's pool-based inference over DART-r, which replicates a plan optimized for a chain of low- and high-class GPUs. Finally, IPIPE consistently achieves high improvements across cluster configurations. For example, compared with NP, IPIPE achieves 67.3%, 34.0%, 47.2%, and 46.6% higher load factors on HC1-L to HC4-L, respectively. This shows IPIPE's robustness on different types of high- and low-class GPUs.

Figure 4.6 shows each framework's temporal utilization of high- and low-class GPUs. While all frameworks achieve high utilization of high-class GPUs, NP and DART-r show zero or low utilization of low-class GPUs. On average, NP, DART-r and IPIPE achieve utilizations

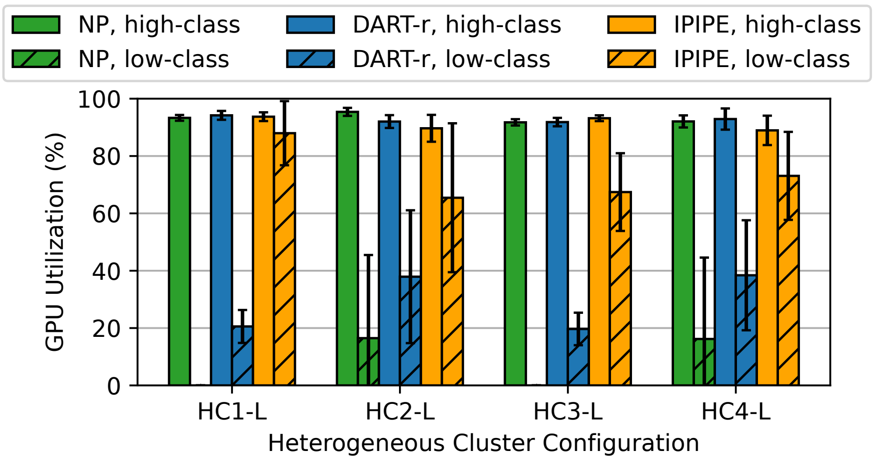


Figure 4.6. GPU temporal utilization under 99% SLO attainment, averaged over DNNs for each cluster configuration.

of 8.2%, 29.1%, and 73.5% respectively on the low-class GPUs. NP's low utilization is caused by the high inference time of whole DNNs on low-class GPUs, which often prohibits low-class GPUs from being used. While DART-r employs DNN partitioning, it chains only one low-end GPU with each high-end GPU, resulting in under-utilization of excess low-end GPUs when their number exceeds that of high-end GPUs.

SLO attainment under varying load factors. Figure 4.7 shows the SLO attainment under varying load factors. For brevity, we showcase only the SLO attainment for DNN group G1, which includes EfficientNet-B8, EncNet, and RtmDet. The attainment is averaged over the 3 DNNs. For example, with IPIPE on HC1-L at 1.0 load factor, the 3 DNNs achieve SLO attainments of 99.4%, 98.6%, and 98.4%, resulting in an averaged attainment of 98.8%.

We observe that IPIPE outperforms both NP and DART-r, achieving the highest SLO attainment under the same load factors. Consequently, it achieves a higher load factor while ensuring 99% SLO attainment. For example, when serving the DNNs on HC1-L, IPIPE ensures 99% SLO attainment under load factors up to 0.95, meaning it can handle at least 95% of the load calculated by the MILP solver. In contrast, NP and DART-r's SLO attainment dips below 99% as the load factor exceeds 0.45 and 0.55, respectively. This is due to the fact that a load factor of 1.0 represents the serving capacity of IPIPE, which is higher than the capacity of NP or DART-r, leading to the dropping of requests beyond their

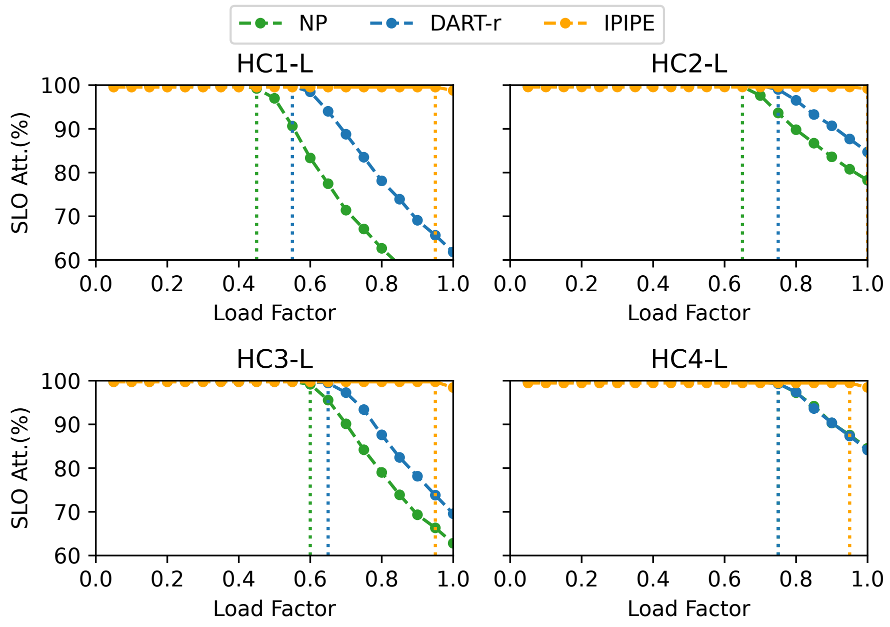


Figure 4.7. SLO attainment of DNN group G1 under varying load factors, averaged over the 3 DNNs in the group. The vertical dotted line denotes the load factor at which each system reaches 99% SLO attainment.

respective serving capacities. Note that although IPIPE achieves higher load factors than the baselines, it may not reach the full load factor of 1.0, as seen with HC1-L, HC2-L, and HC4-L. This is due to the fact that requests arrive in a Poisson arrival, which cannot be fully accounted for by the MILP solver, as discussed in §4.4.

MILP runtime. IPIPE's MILP solver takes 4.6 seconds on average, showing the effectiveness of IPIPE's pre-partitioning (§4.5.3) for reducing the MILP complexity, and enables IPIPE's capability to quickly adapt to dynamic load changes, such as diurnal workload variations or changes in GPU availability within the cluster.

4.7.3 Testbed Results

We verify IPIPE's DNN serving capability with our real implementation on real-world 16-GPU heterogeneous cluster (HC1-S to HC4-S) testbeds deployed on Google Cloud. Due to the testbed's smaller GPU counts, instead of serving DNNs in groups of three as in §4.7.2,

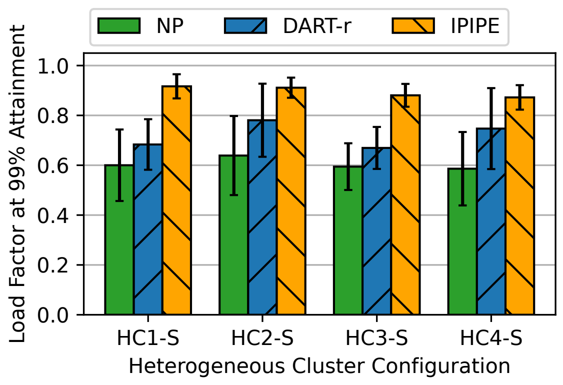


Figure 4.8. The maximum load factor each cluster configuration can achieve under 99% SLO attainment on the testbed, averaged over the DNNs.

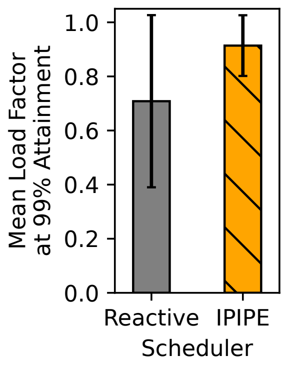


Figure 4.9. SLO attainment with the reactive scheduler and IPIPE's data plane scheduler.

we serve the DNNs one at a time. Figure 4.8 shows the maximum load factor under 99% SLO attainment averaged over the DNNs.

First, we observe that IPIPE consistently outperforms NP, achieving 42.6%–52.8% higher load factors at 99% SLO attainment across the cluster configurations. This validates the advantage of IPIPE's model-parallelism inference in a real-world setting. Secondly, compared to DART-r, IPIPE achieves 16.7%–34.1% higher load factors, which shows the advantage of IPIPE's pool-based inference. Finally, IPIPE continues to achieve consistently high improvements across various cluster configurations. For example, compared with NP, IPIPE achieves 52.8%, 42.6%, 48.1%, and 48.8% higher load factors on HC1-S to HC4-S, respectively. In summary, IPIPE's high improvements over NP and DART-r, previously demonstrated on the discrete-event simulator, remain robust when deployed on a real-world testbed.

4.7.4 Ablation Study: Benefit of Resource Reservation

As discussed in §4.5.5, IPIPE's resource reservation-based data plane dynamically schedules requests onto GPUs to overcome the delays induced by bursty request arrival and network contention. To demonstrate its effectiveness, we compare it to the *reactive* data plane design mentioned in §4.5.2. The reactive scheduler performs adaptive batching independently before each pool of GPUs. It chooses the largest possible batch size, based on the amount of time until the SLO deadline, assuming the resources are available when the batch enters each subsequent stage, *i.e.,* no queuing delay.

Figure 4.9 shows the maximum load factor achieved by the two data plane designs under 99% SLO attainment on HC2-L, averaged over the DNNs. IPIPE achieves an average load factor of 0.91, while reactive only achieves 0.71. The primary factor contributing to this performance degradation, is due to the fact that the reactive scheduler lacks resource usage tracking which leads to batches being scheduled onto servers with saturated network links, causing bloated transfer delays. For example, for EfficientNet-B8, where IPIPE achieves 99% SLO attainment at 1.0 load factor, the reactive scheduler only manages a 0.35 load factor. With 10 Gbps effective bandwidth, feature map transfer between the first and second partition should take 5.1 ms, but the reactive scheduler leads to average transfer times of 18.6 ms and worst-case times of 37.9 ms, resulting in excessive request drops before the second partition in order to meet the SLO.

4.7.5 Microscopic Analysis

In this section, we provide a microscopic analysis of IPIPE, using the FCN model served on the HC3-S cluster on Google Cloud as an example, where IPIPE achieves a load factor of 0.95 under 99% SLO attainment.

Plan structure. Figure 4.10 shows the partitioning plan generated by IPIPE's MILP solver for cluster HC3-S, which consists of 4× V100 and 12× P4 GPUs. The inference latency of the FCN model on the fastest GPU (NVIDIA L4) is 6.66 ms, establishing an SLO of 33.3 ms under an SLO scale of 5. The resulting plan comprises of two pipelines, one with a single partition and the other with two partitions.

The first pipeline consists of a single V100 GPU, performing inference with a batch size of 2, where each batched inference takes 12.3 ms. The theoretical throughput of this pipeline is $(2 \times 1/0.0123) = 162$ requests per second. The second pipeline consists of two partitions, with 12 P4 and 3 V100 GPUs respectively, with 1.4 ms of feature map transfer time in between. Employing batch size unification (§4.5.4), both partitions perform batch

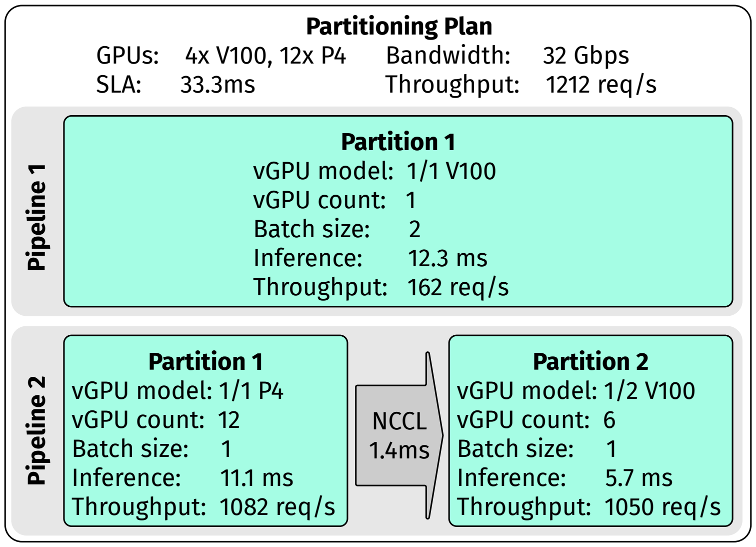


Figure 4.10. The partitioning plan for the FCN model on HC3-S.

size 1 inference. Note that to achieve such a unified batch size, IPIPE subdivides the three V100s in the second partition into six virtual GPUs using NVIDIA MPS. Furthermore, we observe that the two partitions yield similar throughputs of 1082 and 1050 requests per second respectively, resulting in a total throughput of 1050 (the minimum of the two). This shows IPIPE's ability to balance resource allocation between partitions to achieve matched throughput.

Runtime behavior. Figure 4.11 shows an example timeline of the DNN inference on each virtual GPU. We observe that IPIPE performs inference back-to-back in pipeline 1, as well as pipeline 2's partition 2, fully using their GPUs. Note that pipeline 2's partition 1 experiences underutilization, due to the fact that it was provisioned with slightly higher serving throughput than partition 2, as shown in Figure 4.10. Furthermore, the figure showcases that IPIPE's pool-based pipeline allows a batch to be processed by any GPU within each partition, allowing different partitions with different inference latencies to accommodate different numbers of GPUs.

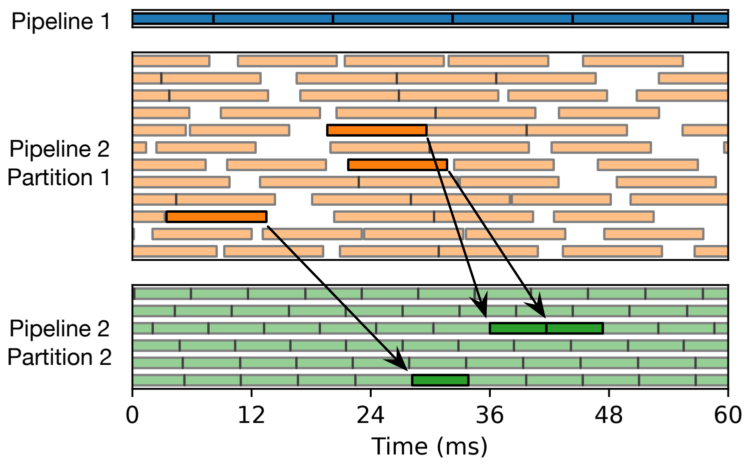
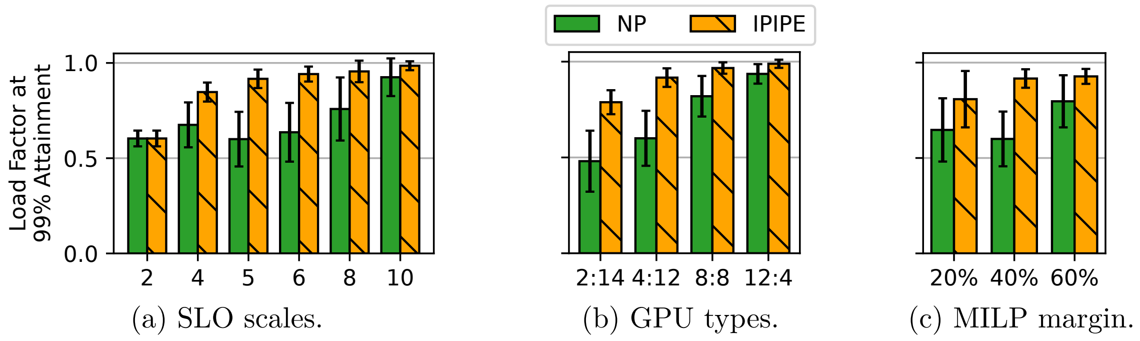


Figure 4.11. An example timeline serving the FCN model on HC3. Each row represents a vGPU, and each box corresponds to a batched inference. The highlighted pairs of boxes denote the same batches across the partitions within a pipeline.

4.7.6 Sensitivity Analysis

In this section, we study IPIPE's sensitivity to various factors, on cluster HC1-S deployed on Google Cloud.



(a) SLO scales. (b) GPU types. (c) MILP margin.

Figure 4.12. Sensitivity of IPIPE to various factors. Results are averaged over 18 DNNs on HC1-S.

Varying SLO scales. In DNN serving settings, operators may exhibit diversity in SLO requirements that deviate from the default SLO scale of 5. We evaluate IPIPE considering various SLO scales ranging from 2 to 10 at intervals of 2, as shown in Figure 4.12a. We observe that IPIPE's performance peaks at a load factor of 5, achieving an improvement of 52.8%, but exhibits a decline with either lower or higher SLO scales.

For example, with an SLO scale of 2, IPIPE shows no improvement over NP, as such stringent SLOs render the utilization of low-class GPUs impractical for either NP or IPIPE. Consequently, IPIPE resorts to running entire DNNs on high-class GPUs, essentially falling back to NP. Conversely, as the SLO scale increases to 10, IPIPE's improvement over NP becomes marginal, due to the fact that more DNNs can now meet the SLO running exclusively on low-class GPUs, and hence the low-class GPUs can be utilized in NP, thereby narrowing the gap between NP and IPIPE.

Varying GPU ratios. In Figure 4.12b, we evaluate IPIPE under varying ratios of high-class (NVIDIA L4) to low-class (NVIDIA P4) GPUs, which shows that IPIPE attains more improvements over NP on clusters with fewer high-class GPUs. For instance, with a high-low ratio of 2:14, IPIPE achieves a 64.03% higher load factor; as the percentage of high-class GPUs increases, IPIPE's improvement diminishes, reaching 5.64% at a high-low ratio of 12:4.

Varying SLO margin size. As discussed in §4.7.1, a 40% margin was subtracted from the SLO in both the MILP formulation of IPIPE and the determination of batch sizes in NP. The impact of the margin size is two-fold — a larger margin size reduces the ideal-case serving capacity (*i.e.,* what load factor 1.0 signifies), but increases the load factor achievable under 99% SLO attainment in practice. Figure 4.12c shows that under varying margin sizes, IPIPE's attainment increases with larger margin sizes, but plateaus as the margin size increases beyond 40%. Furthermore, IPIPE achieves the highest gain of 52.8% over NP under the margin size of 40%, but also maintains a relatively high improvement over NP at 20% and 60% margin sizes, of 24.9% and 16.4% respectively.

4.8 Related Work

Model serving with pipeline parallelism. Several works on serving DNNs explore pipeline parallelism to address a variety of scenarios and objectives. DART [136] partitions DNNs onto a chain of CPUs and GPUs. However, it does not scale to large GPU clusters, as it will generate as many partitions as the number of GPUs in the cluster, resulting in frequent transfer of feature maps which is costly. AlpaServe [129] improves serving throughput by employing pipeline parallelism that facilitates the multiplexing of GPUs across multiple DNNs, but it does not exploit pipeline parallelism on heterogeneous clusters. Another line of works specialize in serving RNN, transformer, or hierarchical DNN models [169]–[171] in a pipelined fashion.

Serving whole DNNs on heterogeneous clusters. Such works typically focus on serving multiple DNNs concurrently, and study the placement of whole DNNs on servers with the goal of either minimizing the cost of cloud VMs [93], [107], [166], or maximizing the throughput of on-premise clusters [105], [167]. Furthermore, several works have studied serving video analytics applications that utilize multiple DNNs forming a Directed Acyclic Graph (DAG) [96], [127], [168]. However, such works do not exploit the diversity across the layers within a DNN.

DNN serving on homogeneous clusters. Several works focus on model serving on homogeneous clusters [28], [34], [103], [128]. However, they do not exploit DNN partitioning or take advantage of the heterogeneity of GPUs.

DNN training with model parallelism. Various works exploit tensor or pipeline parallelism (or both) in model training [132], [135], [137], [138], [172]. Compared to inference, training has a different set of scenarios and requirements, *e.g.,* it does not need to meet SLOs or tackle non-deterministic request arrivals.

4.9 Summary

In this chapter, we presented IPIPE, a system for making effective use of mixed GPUs on heterogeneous clusters in serving video analytics applications. The key innovation of IPIPE is three-fold: pool-based pipelined model inference, an MILP-based control plane that prescribes optimal pipeline plans, and a data plane that performs resource reservation-based adaptive batching to handle runtime dynamics due to asynchronous and bursty request arrivals. Evaluation results on production workloads show that IPIPE achieves 15.9%–64.0% higher serving throughput compared to various baselines.

4.A Mathematical Representation of Basic MILP Formulation

Table 4.3. Inputs to the MILP formulation (top) and values derived from inputs (bottom).

Input	Description
N_k	GPU count of GPU class k
T	The latency SLO
$L_{kb\text{i}}$	The inference latency of layer i under batch size b on GPU class k
S_i	The output feature map size of layer i under batch size 1
D_l	Number of partitions in pipeline l
G_k	A list of tuples (l, d) indicating GPU class k is used for partition d of pipeline l
M	Number of layers in the DNN model
$C_{ldb\text{ij}}$	The inference latency of DNN partition consisting of layers i to j (exclusive) with batch size b on GPU class associated with (l, d)
$X_{ldb\text{ij}}$	The inference throughput of DNN partition consisting of layers i to j (exclusive) with batch size b on a single GPU associated with (l, d)
$Y_{b\text{j}}$	The transfer latency of the feature map of layer j $-$ 1 with batch size b

Table 4.4. Output (top) and intermediate (bottom) decision variables in the MILP formulation.

Variable	Description
$p_{ldb\text{ij}} \in \{0, 1\}$	Whether partition d in pipeline l spans from layer i to j (exclusive) and runs at batch size b
$g_{ldb\text{ij}} \in \mathbb{N}$	Number of GPUs used by partition d in pipeline l
$t_{ld} \in \mathbb{R}_{\geq 0}$	Inference latency of partition d in pipeline l
$x_{ld} \in \mathbb{R}_{\geq 0}$	Inference throughput of partition d in pipeline l
$n_{ld} \in \mathbb{R}_{\geq 0}$	Transfer latency between partition d and partition $d + 1$ in pipeline l
$x_l \in \mathbb{R}_{\geq 0}$	Inference throughput of pipeline l

As shown in Table 4.3, the MILP formulation takes as input the cluster configuration, the latency SLO, and profiling information of the target DNN model. For the convenience of formulating mathematical constraints, the raw inputs are further processed and transformed into different representations. Table 4.4 lists both output decision variables and intermediate decision variables that will only be used inside the MILP formulation. The MILP solution outputs the partition points for each pipeline, as well as the batch size and number of GPUs used by each partition. The MILP formulation maximizes the total inference throughput across all pipelines:

maximize $\sum_l x_l$

The optimization is under the constraint that inference latency of each pipeline does not exceed the latency SLO, and the total number of GPUs allocated across partitions does not exceed the cluster configuration. The constraints are formally formulated below.

$$\sum_{\mathrm{bij}} p_{ld b\mathrm{ij}} = 1 \qquad \forall l, d \tag{4.1}$$

$$p_{ld b\mathrm{ij}} = 0 \qquad \forall l, d, b, \mathrm{i} \geq \mathrm{j} \tag{4.2}$$

$$\sum_{b\mathrm{i}} p_{ld b\mathrm{ij}} = 1 \rightarrow \sum_{b'\mathrm{j}'} p_{ld'b'\mathrm{i}'\mathrm{j}'} = 1 \qquad \forall l, d' = d+1, \mathrm{i}' = \mathrm{j} \tag{4.3}$$

$$\sum_{b\mathrm{j}} p_{ld b\mathrm{ij}} = 1 \qquad \forall l, d = 0, \mathrm{i} = 0 \tag{4.4}$$

$$\sum_{b\mathrm{i}} p_{ld b\mathrm{ij}} = 1 \qquad \forall l, d = D_l - 1, \mathrm{j} = M \tag{4.5}$$

$$p_{ld b\mathrm{ij}} = 0 \rightarrow g_{ld b\mathrm{ij}} = 0 \qquad \forall l, d, b, \mathrm{i}, \mathrm{j} \tag{4.6}$$

$$p_{ld b\mathrm{ij}} = 1 \rightarrow g_{ld b\mathrm{ij}} \geq 1 \qquad \forall l, d, b, \mathrm{i}, \mathrm{j} \tag{4.7}$$

$$\sum_{b\mathrm{ij},(l,d)\in G_k} g_{ld b\mathrm{ij}} \leq N_k \qquad \forall k \tag{4.8}$$

$$t_{ld} = \sum_{b\mathrm{ij}} C_{ld b\mathrm{ij}} \cdot p_{ld b\mathrm{ij}} \qquad \forall l, d \tag{4.9}$$

$$x_{ld} = \sum_{b\mathrm{ij}} X_{ld b\mathrm{ij}} \cdot g_{ld b\mathrm{ij}} \qquad \forall l, d \tag{4.10}$$

$$n_{ld} = \sum_{b\mathrm{ij}} Y_{b\mathrm{j}} \cdot p_{ld b\mathrm{ij}} \qquad \forall l, d \tag{4.11}$$

$$\sum_d t_{ld} + \sum_d n_{ld} \leq T \qquad \forall l \tag{4.12}$$

$$x_l = \min_d x_{ld} \qquad \forall l \tag{4.13}$$

Equations (4.1)–(4.5) ensure DNN partitions are well formed, *i.e.,* partitions cannot be empty, the last and first layers in adjacent partitions must also be adjacent, and the first partition within a pipeline must start with the first layer, while the opposite applies to the last partition. Equation (4.8) represents the constraint on the total GPU count. Equations (4.9)–(4.11) calculates for each partition the inference latency, inference throughput, and transfer latency, respectively. Finally, Equation (4.12) enforces the latency SLO constraint.

The formulation can be easily scaled to the case of multiple DNN models, where each DNN model has its own set of decision variables and constraints, with the additional constraint that the total number of GPUs allocated to all DNN models does not exceed the cluster configuration.

4.B Mathematical Representation of MILP Formulation with Batch Size Unification

Table 4.5. Inputs to the MILP formulation (with batch size unification) (top) and values derived from inputs (bottom).

Input	Description
N_k	GPU count of GPU class k
T	The latency SLO
$L_{kvb\mathrm{i}}$	The inference latency of block i under batch size b on virtual GPU of size $1/v$ and GPU class k
S_i	The output feature map size of block i under batch size 1
D_l	Number of blocks in pipeline l
G_k	A list of tuples (l, d) indicating GPU class k is used for partition d of pipeline l
M	Number of layers in the DNN model
$C_{ldvb\mathrm{ij}}$	The inference latency of DNN partition consisting of blocks i to j (exclusive) with batch size b on $1/v$ virtual GPU of GPU class associated with (l, d)
$X_{ldvb\mathrm{ij}}$	The inference throughput of DNN partition consisting of blocks i to j (exclusive) with batch size b on $1/v$ virtual GPU of GPU class associated with (l, d)
$Y_{b\mathrm{j}}$	The transfer latency of the feature map of block $\mathrm{j}-1$ with batch size b

The inputs and decision variables to the MILP formulation with batch size unification (Table 4.5 and Table 4.6) are similar to that of the basic MILP formulation (§4.A), with the exception that both the model profiling inputs and decision variables now include an additional dimension representing virtual GPUs, and that the profiling inputs are for blocks

Table 4.6. Output (top) and intermediate (bottom) decision variables in the MILP formulation with batch size unification.

Variable	Description
$p_{ldvb\mathrm{ij}} \in \{0,1\}$	Whether partition d in pipeline l spans from block i to j (exclusive) and runs at batch size b on $1/v$ virtual GPU
$g_{ldvb\mathrm{ij}} \in \mathbb{N}$	Number of virtual GPUs used by partition d in pipeline l
$t_{ld} \in \mathbb{R}_{\geq 0}$	Inference latency of partition d in pipeline l
$x_{ld} \in \mathbb{R}_{\geq 0}$	Inference throughput of partition d in pipeline l
$n_{ld} \in \mathbb{R}_{\geq 0}$	Transfer latency between partition d and partition $d+1$ in pipeline l
$x_l \in \mathbb{R}_{\geq 0}$	Inference throughput of pipeline l

instead of layers, and the same applies to the last two dimensions of the output decision variables. The MILP formulation optimizes for the same throughput objective:

$$\text{maximize } \textstyle\sum_l x_l$$

The optimization is under a similar set of constraints as shown below.

$$\textstyle\sum_{v\mathrm{bij}} p_{ldv\mathrm{bij}} = 1 \qquad \forall l, d \tag{4.14}$$

$$p_{ldv\mathrm{bij}} = 0 \qquad \forall l, d, v, b, \mathrm{i} \geq \mathrm{j} \tag{4.15}$$

$$\textstyle\sum_{v\mathrm{i}} p_{ldv\mathrm{bij}} = 1 \rightarrow \sum_{v'\mathrm{j}'} p_{ld'v'\mathrm{bi}'\mathrm{j}'} = 1 \qquad \forall l, b, d' = d+1, \mathrm{i}' = \mathrm{j} \tag{4.16}$$

$$\textstyle\sum_{v\mathrm{bj}} p_{ldv\mathrm{bij}} = 1 \qquad \forall l, d = 0, \mathrm{i} = 0 \tag{4.17}$$

$$\textstyle\sum_{v\mathrm{bi}} p_{ldv\mathrm{bij}} = 1 \qquad \forall l, d = D_l - 1, \mathrm{j} = M \tag{4.18}$$

$$p_{ldv\mathrm{bij}} = 0 \rightarrow g_{ldv\mathrm{bij}} = 0 \qquad \forall l, d, v, b, \mathrm{i}, \mathrm{j} \tag{4.19}$$

$$p_{ldv\mathrm{bij}} = 1 \rightarrow g_{ldv\mathrm{bij}} \geq 1 \qquad \forall l, d, v, b, \mathrm{i}, \mathrm{j} \tag{4.20}$$

$$\textstyle\sum_{v\mathrm{bij},(l,d)\in G_k} g_{ldv\mathrm{bij}}/v \leq N_k \qquad \forall k \tag{4.21}$$

$$t_{ld} = \textstyle\sum_{v\mathrm{bij}} C_{ldv\mathrm{bij}} \cdot p_{ldv\mathrm{bij}} \qquad \forall l, d \tag{4.22}$$

$$x_{ld} = \textstyle\sum_{v\mathrm{bij}} X_{ldv\mathrm{bij}} \cdot g_{ldv\mathrm{bij}} \qquad \forall l, d \tag{4.23}$$

$$n_{ld} = \textstyle\sum_{v\mathrm{bij}} Y_{\mathrm{bj}} \cdot p_{ldv\mathrm{bij}} \qquad \forall l, d \tag{4.24}$$

$$\textstyle\sum_d t_{ld} + \sum_d n_{ld} \leq T \qquad \forall l \tag{4.25}$$

$$x_l = \min_d x_{ld} \qquad \forall l \tag{4.26}$$

The major differences lie in Equations (4.16) and (4.21). In Equations (4.16), the dimension b is not summed over, which enforces that the same batch size used by the first partition will also need to be used by the next partition. In Equations (4.21), the decision variable g represents the count of *virtual* GPUs, thus, dividing it by v gives us the number of *physical* GPUs that the partition uses (note that v is not a decision variable and such divisions are allowed in MILP).

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Achieving high accuracy and efficient AR task offloading in real-world AR app deployment faces two fundamental challenges: the need to offload multiple DNN-supported tasks concurrently to achieve app functionality, and the necessity for concurrent AR clients from a large user base to offload to a cluster of GPU servers. Addressing these challenges requires scaling existing systems to handle an increasing number of AR tasks, AR clients, and edge servers, which in turn prompts new design goals and introduces complex scheduling challenges.

In this thesis, we have presented systems designed to address these unique challenges arising from scaling AR tasks, AR clients, and edge servers, respectively. Firstly, we presented AccuMO, the first framework that dynamically schedules offloading of multiple compute-intensive DNN tasks of an AR app while optimizing the overall DNN inference accuracy across the tasks. We employ a two-level control feedback loop that adapts between alternative local execution options within each task module and globally balancing offloading among multiple tasks using MPC. AccuMO improves the overall task accuracy by on average 7.6%–14.3% over the best baseline under different accuracy weight ratios for depth estimation and odometry. Secondly, we presented ARISE, the first framework that addresses the AR inference serving problem for individual edge servers. The framework employs an online accuracy estimator that estimates the accuracy for each AR client under various configurations and an online scheduler that proactively coordinate requests from the clients served by a server. ARISE supports 1.7x–6.9x more clients compared to various baselines while keeping the per-client accuracy drops with the client-specified SLA. Lastly, we presented IPIPE, a system for making effective use of mixed GPUs on heterogeneous clusters in serving video analytics applications in general. The key innovation of IPIPE is three-fold: pool-based pipelined model inference, an MILP-based control plane that prescribes optimal pipeline plans, and a data plane that performs resource reservation-based adaptive batching to handle runtime dynamics due to asynchronous and bursty request arrivals. IPIPE achieves 15.9%–64.0% higher serving throughput compared to various baselines.

5.2 Future Work

Holistic edge-assisted AR system design. In this thesis, we presented systems that handle the scaling of AR tasks, AR clients, and edge servers. However, each system only tackles one aspect, without consideration of the others. Simply stapling the systems together may result in conflicts in scheduling decisions and lead to suboptimal schedules. With multiple AR tasks and multiple edge servers, the per-task offloading frequency and the task-server assignment may interact with each other. For example, increasing the offloading frequency for one task to attain higher task accuracy would require allocating more servers to that task to maintain the current client count. An important step forward in the design of edge-assisted AR systems is to integrate all three aspects into a single, cohesive system that manages such complex interactions effectively. Furthermore, such a holistic design unlocks more optimization opportunities. For instance, consider multiple AR clients with different content accuracy requirements offloading to a cluster of edge servers. Assigning clients with similar accuracy drop rates to the same server can improve serving capacity by enabling larger batch sizes for servers handling clients with lower accuracy drop rates.

Edge-assisted AR system and network co-design. Due to the stringent latency requirements, edge-assisted AR is widely regarded as a "killer" app for 5G, which supports ultra-high bandwidth and low latency. However, as the number of AR clients increases, network congestion may still occur, leading to longer frame transfer latency and reduced AR task accuracy due to shared uplink channels. AR system-aware network scheduling, *e.g.,* prioritizing requests with higher accuracy requirements while considering the impact on batched inference due to changes in request arrival, can enhance overall task accuracy. Frame transfer latency can be further reduced by making the uplink resource grant mechanism AR system-aware. Traditionally, the base station grants uplink resources only after clients have communicated their transmission needs, which occurs after frames have been pushed into the client-side transmission buffer, adding extra round-trips to the critical path. However, an AR system-aware design can schedule and grant uplink resources in advance based on expected frame transfer timings and sizes, streamlining the frame transfer process.

Generalization to system design in other domains. The systems presented in this thesis are specifically designed for supporting edge-assisted AR systems by leveraging unique AR workload characteristics, with some systems, such as IPIPE, being broadly applicable to video analytics systems. Generally, these workloads are restricted to computer vision tasks. However, the insights and techniques developed can potentially be applied to other domains. For example, in natural language processing, large language models (LLMs) consist of uniform layers, which precludes the use of our insights on non-uniform layers. Nevertheless, we can apply the technique of pairing low-end GPUs with high-end GPUs to utilize low-end GPUs when running the entire model on low-end GPUs exceeds the latency SLO. Additionally, language models exhibit diversity in computational characteristics between the prefill and decode stages. Pipelining these stages can also leverage different GPU types, thus optimizing performance across various computational tasks.

REFERENCES

[1] MarketsandMarkets. "Augmented reality (ar) market." (Aug. 2021), [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/augmented-reality-market-82758548.html.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[3] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[4] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, 2016, pp. 379–387.

[5] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[6] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[7] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[8] "Depth adds realism." (2022), [Online]. Available: https://developers.google.com/ar/develop/depth.

[9] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom 19, Los Cabos, Mexico: Association for Computing Machinery, 2019, ISBN: 9781450361699. DOI: 10.1145/3300061.3300116. [Online]. Available: https://doi.org/10.1145/3300061.3300116.

[10] J. Meng, Z. Kong, Q. Xu, and Y. C. Hu, "Do larger (more accurate) deep neural network models help in edge-assisted augmented reality?" In *Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration*, ser. NAI'21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 47–52, ISBN: 9781450386333. DOI: 10.1145/3472727.3472807. [Online]. Available: https://doi.org/10.1145/3472727.3472807.

[11] R. Xu, J. Lee, P. Wang, S. Bagchi, Y. Li, and S. Chaterji, "Litereconfig: Cost and content aware reconfiguration of video object detection systems for mobile gpus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, Rennes, France: Association for Computing Machinery, 2022, pp. 334–351, ISBN: 9781450391627. DOI: 10.1145/3492321.3519577. [Online]. Available: https://doi.org/10.1145/3492321.3519577.

[12] M. Ghoshal, P. Dash, Z. Kong, *et al.*, "Can 5g mmwave support multi-user ar?" In *Passive and Active Measurement*, O. Hohlfeld, G. Moura, and C. Pelsser, Eds., Cham: Springer International Publishing, 2022, pp. 180–196, ISBN: 978-3-030-98785-5.

[13] M. Almeida, S. Laskaridis, A. Mehrotra, L. Dudziak, I. Leontiadis, and N. D. Lane, "Smart at what cost? characterising mobile deep neural networks in the wild," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, pp. 658–672, ISBN: 9781450391290. DOI: 10.1145/3487552.3487863. [Online]. Available: https://doi.org/10.1145/3487552.3487863.

[14] J. Mao, Q. Yang, A. Li, H. Li, and Y. Chen, "Mobieye: An efficient cloud-based video detection system for real-time mobile applications," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, Las Vegas, NV, USA: Association for Computing Machinery, 2019, ISBN: 9781450367257. DOI: 10.1145/3316781.3317865. [Online]. Available: https://doi.org/10.1145/3316781.3317865.

[15] *AWS Wavelength - Deliver ultra-low latency applications for 5G devices*, Online. [Online]. Available: https://aws.amazon.com/wavelength/.

[16] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015, pp. 155–168.

[17] Y. Kang, J. Hauswald, C. Gao, *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[18] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: Synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–15.

[19] S. Yao, J. Li, D. Liu, *et al.*, "Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 476–488.

[20] J. Simon. "Amazon elastic inference gpu-powered deep learning inference acceleration." (Nov. 2018), [Online]. Available: https://aws.amazon.com/blogs/aws/amazon-elastic-inference-gpu-powered-deep-learning-inference-acceleration/.

[21] *Cloud-native computing platform*, https://puzl.ee/resources/gpu, 2019.

[22] S. Chandrasekaran. "Ride the fast lane to ai productivity with multi-instance gpus." (May 2020), [Online]. Available: https://blogs.nvidia.com/blog/2020/05/14/multi-instance-gpus/.

[23] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *Proc. of IEEE ICMLA*, 2015.

[24] Q. Weng, W. Xiao, Y. Yu, *et al.*, "{Mlaas} in the wild: Workload analysis and scheduling in {large-scale} heterogeneous {gpu} clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.

[25] F. Heinrich, L. Schwenderling, F. Joeres, K. Lawonn, and C. Hansen, "Comparison of augmented reality display techniques to support medical needle insertion," *Proc. of IEEE TVCG*, vol. 26, no. 12, pp. 3568–3575, 2020.

[26] R. Xu, C.-l. Zhang, P. Wang, *et al.*, "Approxdet: Content and contention-aware approximate object detection for mobiles," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, ser. SenSys '20, Virtual Event, Japan: Association for Computing Machinery, 2020, pp. 449–462, ISBN: 9781450375900. DOI: 10.1145/3384419.3431159. [Online]. Available: https://doi.org/10.1145/3384419.3431159.

[27] E. Camacho and C. Alba, *Model Predictive Control* (Advanced Textbooks in Control and Signal Processing). Springer London, 2013, ISBN: 9780857293985. [Online]. Available: https://books.google.com/books?id=tXZDAAAAQBAJ.

[28] H. Shen, L. Chen, Y. Jin, *et al.*, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 322–337, ISBN: 9781450368735. DOI: 10.1145/3341301.3359658. [Online]. Available: https://doi.org/10.1145/3341301.3359658.

[29] Z. J. Kong, Q. Xu, J. Meng, and Y. C. Hu, “Accumo: Accuracy-centric multitask offloading in edge-assisted mobile augmented reality,” in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ser. ACM MobiCom ’23, Madrid, Spain: Association for Computing Machinery, 2023, ISBN: 9781450399906. DOI: 10.1145/3570361.3592531. [Online]. Available: https://doi.org/10.1145/3570361.3592531.

[30] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali, “Reducto: On-camera filtering for resource-efficient real-time video analytics,” in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 359–376.

[31] K. Chen, T. Li, H.-S. Kim, D. E. Culler, and R. H. Katz, “Marvel: Enabling mobile augmented reality with low energy and low latency,” in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys ’18, Shenzhen, China: Association for Computing Machinery, 2018, pp. 292–304, ISBN: 9781450359528. DOI: 10.1145/3274783.3274834. [Online]. Available: https://doi.org/10.1145/3274783.3274834.

[32] H. Zhou, S. Bateni, and C. Liu, “S^3DNN: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 190–201. DOI: 10.1109/RTAS.2018.00028.

[33] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, “Pretzel: Opening the black box of machine learning prediction serving systems,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 611–626, ISBN: 9781931971478.

[34] A. Gujarati, R. Karimi, S. Alzayat, *et al.*, “Serving dnns like clockwork: Performance predictability from the bottom up,” in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’20, USA: USENIX Association, 2020, ISBN: 978-1-939133-19-9.

[35] F. Yu, S. Bray, D. Wang, *et al.*, “Automated runtime-aware scheduling for multi-tenant dnn inference on gpu,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643501.

[36] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 199–216, ISBN: 978-1-939133-29-53. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/choi-seungbeom.

[37] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, *Detectron2*, https://github.com/facebookresearch/detectron2, 2019.

[38] R. Liu, L. Zhang, J. Wang, H. Yang, and Y. Liu, "Petri: Reducing bandwidth requirement in smart surveillance by edge-cloud collaborative adaptive frame clustering and pipelined bidirectional tracking," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 421–426. DOI: 10.1109/DAC18074.2021.9586088.

[39] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury, "Frugal following: Power thrifty object detection and tracking for mobile augmented reality," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 96–109.

[40] H. Yeo, C. J. Chong, Y. Jung, J. Ye, and D. Han, "Nemo: Enabling neural-enhanced video streaming on commodity mobile devices," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20, London, United Kingdom: Association for Computing Machinery, 2020, ISBN: 9781450370851. DOI: 10.1145/3372224.3419185. [Online]. Available: https://doi.org/10.1145/3372224.3419185.

[41] C. Gao, Y. Wang, W. Chen, and L. Zhang, "An intelligent video processing architecture for edge-cloud video streaming," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 415–420. DOI: 10.1109/DAC18074.2021.9586328.

[42] C.-H. Yao, C. Fang, X. Shen, Y. Wan, and M.-H. Yang, "Video object detection via object-level temporal aggregation," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., Cham: Springer International Publishing, 2020, pp. 160–177, ISBN: 978-3-030-58568-6.

[43] J. Zhang, D. Zhang, X. Xu, *et al.*, "Mobipose: Real-time multi-person pose estimation on mobile devices," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, ser. SenSys '20, Virtual Event, Japan: Association for Computing Machinery, 2020, pp. 136–149, ISBN: 9781450375900. DOI: 10.1145/3384419.3430726. [Online]. Available: https://doi.org/10.1145/3384419.3430726.

[44] M. G. Yong. "Object detection and tracking using mediapipe." (Dec. 2019), [Online]. Available: https://developers.googleblog.com/2019/12/object-detection-and-tracking-using-mediapipe.html.

[45] A. Ahmadyan and T. Hou. "Real-time 3d object detection on mobile devices with mediapipe." (Mar. 2020), [Online]. Available: https://ai.googleblog.com/2020/03/real-time-3d-object-detection-on-mobile.html.

[46] S. F. Bhat, I. Alhashim, and P. Wonka, "Adabins: Depth estimation using adaptive bins," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 4009–4018.

[47] X.-Y. Kuo, C. Liu, K.-C. Lin, E. Luo, Y.-W. Chen, and C.-Y. Lee, "Dynamic attention-based visual odometry," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 5753–5760. DOI: 10.1109/IROS45743.2020.9340890.

[48] X. Xie and K.-H. Kim, "Source compression with bounded dnn perception loss for iot edge computer vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19, Los Cabos, Mexico: Association for Computing Machinery, 2019, ISBN: 9781450361699. DOI: 10.1145/3300061.3345448. [Online]. Available: https://doi.org/10.1145/3300061.3345448.

[49] K. Du, A. Pervaiz, X. Yuan, *et al.*, "Server-driven video streaming for deep learning inference," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 557–570.

[50] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 1423–1431.

[51] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *2018 IEEE 24th international conference on parallel and distributed systems (ICPADS)*, IEEE, 2018, pp. 671–678.

[52] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, 2019.

[53] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, 2018, pp. 31–36.

[54] A. Banitalebi-Dehkordi, N. Vedula, J. Pei, F. Xia, L. Wang, and Y. Zhang, "Auto-split: A general framework of collaborative edge-cloud ai," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 2543–2553, ISBN: 9781450383325. DOI: 10.1145/3447548.3467078. [Online]. Available: https://doi.org/10.1145/3447548.3467078.

[55] W. Zhang, Z. He, L. Liu, *et al.*, "Elf: Accelerate high-resolution mobile deep vision with content-aware parallel offloading," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 201–214.

[56] P. Guo, B. Hu, R. Li, and W. Hu, "Foggycache: Cross-device approximate computation reuse," in *Proceedings of the 24th annual international conference on mobile computing and networking*, 2018, pp. 19–34.

[57] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, IEEE, 2017, pp. 276–286.

[58] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 123–136.

[59] B. Hu and W. Hu, "Linkshare: Device-centric control for concurrent and continuous mobile-cloud interactions," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 15–29.

[60] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 271–284.

[61] J. Yi and Y. Lee, "Heimdall: Mobile gpu coordination platform for augmented reality applications," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[62] N. Ling, K. Wang, Y. He, G. Xing, and D. Xie, "Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 1–14.

[63] D. Wofk, F. Ma, T.-J. Yang, S. Karaman, and V. Sze, "Fastdepth: Fast monocular depth estimation on embedded systems," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 6101–6108.

[64] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 236–252.

[65] 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*, ser. SIGCOMM '15, London, United Kingdom: Association for Computing Machinery, 2015, pp. 325–338, ISBN: 9781450335423. DOI: 10.1145/2785956.2787486. [Online]. Available: https://doi.org/10.1145/2785956.2787486.

[66] J. Watson, O. Mac Aodha, V. Prisacariu, G. Brostow, and M. Firman, "The temporal opportunist: Self-supervised multi-frame monocular depth," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 1164–1174.

[67] M. Ramamonjisoa, M. Firman, J. Watson, V. Lepetit, and D. Turmukhambetov, "Single image depth prediction with wavelet decomposition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 11 089–11 098.

[68] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer," *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[69] J. Solà, *Quaternion kinematics for the error-state kalman filter*, 2017. DOI: 10.48550/ARXIV.1711.02508. [Online]. Available: https://arxiv.org/abs/1711.02508.

[70] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361. DOI: 10.1109/CVPR.2012.6248074.

[71] H. Zhan, C. S. Weerasekera, J.-W. Bian, and I. Reid, "Visual odometry revisited: What should be learnt?" In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 4203–4210. DOI: 10.1109/ICRA40945.2020.9197374.

[72] D. Birkes and Y. Dodge, *Alternative Methods of Regression* (Wiley Series in Probability and Statistics). Wiley, 2011, ISBN: 9781118150245. [Online]. Available: https://books.google.com/books?id=CIedErj0HKcC.

[73] *Ncnn: A high-performance neural network inference computing framework optimized for mobile platforms*, https://github.com/Tencent/ncnn, Shenzhen, China: Tencent, 2022.

[74] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, "Flownet 2.0: Evolution of optical flow estimation with deep networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017.

[75] Google Brain Team, *Tensorflow lite*, https://www.tensorflow.org/lite, Mountain View, USA, 2022.

[76] *Sensor fusion and tracking toolbox*, https://www.mathworks.com/products/sensor-fusion-and-tracking.html, Natick, USA: MathWorks, 2022.

[77] *Matlab coder*, https://www.mathworks.com/products/matlab-coder.html, Natick, USA: MathWorks, 2022.

[78] S. Jha, Y. Li, S. Noghabi, *et al.*, "Visage: Enabling timely analytics for drone imagery," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '21, New Orleans, Louisiana: Association for Computing Machinery, 2021, pp. 789–803, ISBN: 9781450383424. DOI: 10.1145/3447993.3483273. [Online]. Available: https://doi.org/10.1145/3447993.3483273.

[79] S. Shi, J. Cui, Z. Jiang, *et al.*, "Vips: Real-time perception fusion for infrastructure-assisted autonomous driving," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, ser. MobiCom '22, Sydney, NSW, Australia: Association for Computing Machinery, 2022, pp. 133–146, ISBN: 9781450391818. DOI: 10.1145/3495243.3560539. [Online]. Available: https://doi.org/10.1145/3495243.3560539.

[80] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[81] "Inertial measurement unit bmi263." (2022).

[82] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "Segformer: Simple and efficient design for semantic segmentation with transformers," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., 2021, pp. 12 077–12 090. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/64f1f27bf1b4ec22924fd0acb550c235-Paper.pdf.

[83] C. Cadena, Y. Latif, and I. D. Reid, "Measuring the performance of single image depth estimation methods," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 4150–4157. DOI: 10.1109/IROS.2016.7759611.

[84] Z. J. Kong, Q. Xu, and Y. C. Hu, "Arise: An accuracy-aware proactive framework for serving concurrent edge-assisted ar clients," in *Proceedings of the 22nd ACM International Conference on Mobile Systems, Applications, and Services*, ser. MOBISYS '24, Minato-ku, Tokyo, Japan: Association for Computing Machinery, 2024, ISBN: 9798400705816. DOI: 10.1145/3643832.3661894. [Online]. Available: https://doi.org/10.1145/3643832.3661894.

[85] Y. Zhang, T. Scargill, A. Vaishnav, G. Premsankar, M. Di Francesco, and M. Gorlatova, "Indepth: Real-time depth inpainting for mobile augmented reality," in *Proc. ACM IMWUT*, 2022.

[86] A. J. Ben Ali, Z. S. Hashemifar, and K. Dantu, "Edge-slam: Edge-assisted visual simultaneous localization and mapping," 2020.

[87] I. Alhashim and P. Wonka, "High quality monocular depth estimation via transfer learning," *arXiv preprint arXiv:1812.11941*, 2018.

[88] A. Howard, M. Sandler, G. Chu, *et al.*, "Searching for mobilenetv3," in *Proc. of IEEE/CVF ICCV*, 2019, pp. 1314–1324.

[89] Y. Zhang, P. Sun, Y. Jiang, *et al.*, "Bytetrack: Multi-object tracking byăassociating every detection box," in *Proc. of ECCV*, 2022.

[90] P. Dash, Z. J. Kong, Y. C. Hu, *et al.*, "How to pipeline frame transfer and server inference in edge-assisted ar to optimize ar task accuracy?" In *Proc. of ACM EdgeSys*, 2023, pp. 36–41.

[91] I. Amazon Web Services. "Amazon rekognition: Automate and lower the cost of your image recognition and video analysis with machine learning." (2024), [Online]. Available: https://aws.amazon.com/rekognition/.

[92] Microsoft. "Azure ai vision." (2024), [Online]. Available: https://azure.microsoft.com/en-us/products/ai-services/ai-%20vision/.

[93] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

[94] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, *et al.*, "Ekya: Continuous learning of video analytics models on edge compute servers," in *Proc. of USENIX NSDI*, 2022.

[95] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *Proceedings of the 2018 conference of the ACM special interest group on data communication*, 2018, pp. 253–266.

[96] Y. Hu, R. Ghosh, and R. Govindan, "Scrooge: A cost-effective deep learning inference system," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21, Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 624–638, ISBN: 9781450386388. DOI: 10.1145/3472883.3486993. [Online]. Available: https://doi.org/10.1145/3472883.3486993.

[97] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. of VLDB*, 2017.

[98] M. Khani, G. Ananthanarayanan, K. Hsieh, *et al.*, "Recl: Responsive resource-efficient continuous learning for video analytics," in *Proc. of USENIX NSDI*, 2023.

[99] S. Paul, K. Rao, G. Coviello, *et al.*, "Enhancing video analytics accuracy via real-time automated camera parameter tuning," in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '22, <conf-loc>, <city>Boston</city>, <state>Massachusetts</state>, </conf-loc>: Association for Computing Machinery, 2023, pp. 291–304, ISBN: 9781450398862. DOI: 10.1145/3560905.3568527. [Online]. Available: https://doi.org/10.1145/3560905.3568527.

[100] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and Delay-Tolerance," in *Proc. of USENIX NSDI*, 2017.

[101] Zhang, Wuyang and He, Zhezhi and Liu, Luyang and Jia, Zhenhua and Liu, Yunxin and Gruteser, Marco and Raychaudhuri, Dipankar and Zhang, Yanyong, "ELF: Accelerate High-resolution Mobile Deep Vision with Content-aware Parallel Offloading," in *Proc. of ACM MobiCom*, 2021.

[102] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00073.

[103] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 613–627, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[104] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: Controlled spatial sharing of gpus for a scalable inference platform," in *Proc. of ACM SoCC*, 2020.

[105] L. Ke, U. Gupta, M. Hempstead, C.-J. Wu, H.-H. S. Lee, and X. Zhang, "Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 141–154. DOI: 10.1109/HPCA53966.2022.00019.

[106] H. Qin, S. Zawad, Y. Zhou, L. Yang, D. Zhao, and F. Yan, "Swift machine learning model serving scheduling: A region based reinforcement learning approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356164. [Online]. Available: https://doi.org/10.1145/3295500.3356164.

[107] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 397–411, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero.

[108] F. Yan, O. Ruwase, Y. He, and E. Smirni, "Serf: Efficient scheduling for fast deep neural network serving via judicious parallelism," in *Proc. of SC*, 2016.

[109] J. Meng, Z. J. Kong, Y. C. Hu, M. G. Choi, and D. Lal, "Do we need sophisticated system design for edge-assisted augented reality?" In *Proc. of ACM EdgeSys*, 2022.

[110] *Nvidia tensorrt*, NVIDIA Corporation, 2024. [Online]. Available: https://developer.nvidia.com/tensorrt.

[111] *Nvjpeg libraries*, NVIDIA Corporation, 2024. [Online]. Available: https://developer.nvidia.com/nvjpeg.

[112] M. Ghoshal *et al.*, "Can 5G mmWave support Multi-User AR?" In *Proc. of PAM*, 2022.

[113] J. Bian, Z. Li, N. Wang, *et al.*, "Unsupervised scale-consistent depth and ego-motion learning from monocular video," *Proc. of NeurIPS*, vol. 32, 2019.

[114] *Tensorflow android camera demo*, 2020. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/%2048a2944c94b190434418d5a7c7f0df452c3aded5/tensorflow/examples/%20android.

[115] A. Milan *et al.*, "MOT16: A Benchmark for Multi-Object Tracking," *arXiv preprint arXiv:1603.00831*, 2016.

[116] N. Agarwal and R. Netravali, "Boggart: Towards general-purpose acceleration of retrospective video analytics," in *Proc. of USENIX NSDI*, 2023.

[117] S. Jiang, Z. Lin, Y. Li, Y. Shu, and Y. Liu, "Flexible high-resolution object detection on edge devices with tunable latency," in *Proc. of ACM MobiCom*, ser. MobiCom '21, New York, NY, USA: Association for Computing Machinery, 2021, pp. 559–572, ISBN: 9781450383424. DOI: 10.1145/3447993.3483274. [Online]. Available: https://doi.org/10.1145/3447993.3483274.

[118] Z. Liu, G. Lan, J. Stojkovic, Y. Zhang, C. Joe-Wong, and M. Gorlatova, "Collabar: Edge-assisted collaborative image recognition for mobile augmented reality," in *Proc. of ACM/IEEE IPSN*, 2020.

[119] P. Ren *et al.*, "Edge AR X5: An Edge-Assisted Multi-User Collaborative Framework for Mobile Web Augmented Reality in 5G and Beyond," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[120] W. Zhang, B. Han, and P. Hui, "Sear: Scaling experiences in multi-user augmented reality," *Proc. of IEEE TVCG*, vol. 28, no. 5, pp. 1982–1992, 2022.

[121] P. Shankar, T. Nadeem, J. Rosca, and L. Iftode, "CARS: Context-Aware Rate Selection for Vehicular Networks," in *Proc. of IEEE ICNP*, 2008.

[122] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[123] *This is the most heavily surveilled city in the US: 50 CCTV cameras per 1,000 citizens*, https://cybernews.com/editorial/this-is-the-most-heavily-surveilled-city-in-the-us-50-cctv-cameras-per-1000-citizens/, Last accessed, April 1, 2024.

[124] *One Legacy of Tiananmen: Chinas 100 Million Surveillance Cameras*, https://www.wsj.com/articles/BL-CJB-22562, Last accessed, December 1, 2021.

[125] *One Surveillance Camera for Every 11 People in Britain, Says CCTV Survey*, https://www.telegraph.co.uk/technology/10172298/One-surveillance-camera-for-every-11-people-in-Britain-says-CCTV-survey.html, Last accessed, December 1, 2024.

[126] *The most surveilled cities in the world*, https://www.usnews.com/news/cities/articles/2020-08-14/the-top-10-most-surveilled-cities-in-the-world, Last accessed, April 1, 2024.

[127] D. Crankshaw, G.-E. Sela, X. Mo, *et al.*, "Inferline: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 477–491, ISBN: 9781450381376. DOI: 10.1145/3419111.3421285. [Online]. Available: https://doi.org/10.1145/3419111.3421285.

[128] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the wild," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 787–808, ISBN: 978-1-939133-33-5. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

[129] Z. Li, L. Zheng, Y. Zhong, *et al.*, “AlpaServe: Statistical multiplexing with model parallelism for deep learning serving,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA: USENIX Association, Jul. 2023, pp. 663–679, ISBN: 978-1-939133-34-2. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

[130] *NVIDIA Hopper Architecture*, https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/, 2023.

[131] *Azure Cognitive Service for Vision*, https://azure.microsoft.com/en-us/products/cognitive-services/vision-services/.

[132] J. H. Park, G. Yun, M. Y. Chang, *et al.*, “Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 307–321.

[133] *Google Distributed Cloud Edge*, https://cloud.google.com/distributed-cloud/edge/latest/docs/gpu, 2023.

[134] *Azure Private Multi-access Edge Compute (MEC)*, https://azure.microsoft.com/en-us/solutions/private-multi-access-edge-compute-mec, 2023.

[135] J. K. Kim, Q. Ho, S. Lee, *et al.*, “Strads: A distributed framework for scheduled model parallel machine learning,” in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[136] Y. Xiang and H. Kim, “Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference,” in *2019 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2019, pp. 392–405.

[137] Y. Huang, Y. Cheng, A. Bapna, *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” vol. 32, 2019.

[138] D. Narayanan, A. Harlap, A. Phanishayee, *et al.*, “Pipedream: Generalized pipeline parallelism for dnn training,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[139] N. Corporation, *NVIDIA TensorRT*, https://developer.nvidia.com/tensorrt, Santa Clara, California, U.S., 2017.

[140] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Sep. 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html.

[141] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[142] *GPU Performance Background User's Guide*, https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html, 2023.

[143] *Matrix Multiplication Background User's Guide*, https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2023.

[144] Gurobi Optimization, LLC., *Gurobi Optimizer*, https://www.gurobi.com/solutions/gurobi-optimizer/, Beaverton, Oregon, U.S., 2008.

[145] *Multi-Process Service*, https://docs.nvidia.com/deploy/mps/index.html, 2023.

[146] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 11 976–11 986.

[147] S. Zhang, C. Chi, Y. Yao, Z. Lei, and S. Z. Li, "Bridging the gap between anchor-based and anchor-free detection via adaptive training sample selection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 9759–9768.

[148] J. He, Z. Deng, L. Zhou, Y. Wang, and Y. Qiao, "Adaptive pyramid context network for semantic segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7519–7528.

[149] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III 14*, Springer, 2016, pp. 649–666.

[150] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, and Q. Tian, "Centernet: Keypoint triplets for object detection," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6569–6578.

[151] M. Yin, Z. Yao, Y. Cao, *et al.*, "Disentangled non-local neural networks," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XV 16*, Springer, 2020, pp. 191–207.

[152] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[153] C. Zhu, Y. He, and M. Savvides, "Feature selective anchor-free module for single-shot object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 840–849.

[154] H. Zhang, K. Dana, J. Shi, *et al.*, "Context encoding for semantic segmentation," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2018, pp. 7151–7160.

[155] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "Repvgg: Making vgg-style convnets great again," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 13 733–13 742.

[156] X. Li, W. Wang, L. Wu, *et al.*, "Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 002–21 012, 2020.

[157] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[158] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[159] C. Lyu, W. Zhang, H. Huang, *et al.*, "Rtmdet: An empirical study of designing real-time object detectors," *arXiv preprint arXiv:2212.07784*, 2022.

[160] Y. Cao, J. Xu, S. Lin, F. Wei, and H. Hu, "Gcnet: Non-local networks meet squeeze-excitation networks and beyond," in *Proceedings of the IEEE/CVF international conference on computer vision workshops*, 2019, pp. 0–0.

[161] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 781–10 790.

[162] X. Wang, R. Girshick, A. Gupta, and K. He, "Non-local neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7794–7803.

[163] *Torchvision Models and Pretrained Weights*, https://pytorch.org/vision/stable/models.html.

[164] *OpenMMLab Open Platform*, https://platform.openmmlab.com/home/.

[165] *OpenVINO Model Zoo*, https://docs.openvino.ai/2023.2/model_zoo.html.

[166] B. Li, R. B. Roy, T. Patel, V. Gadepally, K. Gettings, and D. Tiwari, "Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021, ISBN: 9781450384421. DOI: 10.1145/3458817.3476168. [Online]. Available: https://doi.org/10.1145/3458817.3476168.

[167] B. Li, S. Samsi, V. Gadepally, and D. Tiwari, "Kairos: Building cost-efficient machine learning inference systems with heterogeneous cloud resources," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 3–16.

[168] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[169] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "Turbotransformers: An efficient gpu serving system for transformer models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 389–402.

[170] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {transformer-based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[171] A. Goel, C. Tung, X. Hu, G. K. Thiruvathukal, J. C. Davis, and Y.-H. Lu, "Efficient computer vision on edge devices with pipeline-parallel hierarchical neural networks," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 532–537. DOI: 10.1109/ASP-DAC52403.2022.9712574.

[172] L. Zheng, Z. Li, H. Zhang, *et al.*, “Alpa: Automating inter-and {intra-operator} parallelism for distributed deep learning,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 559–578.

VITA

Qiang Xu is a Ph.D. student in Elmore Family School of Electrical and Computer Engineering at Purdue University, supervised by Prof. Y. Charlie Hu. He is interested in mobile systems, edge computing, and their intersection with machine learning. His recent research focuses on building efficient machine learning inference systems for emerging applications like augmented reality. He received his B.E. in Computer Science and Technology from University of Science and Technology of China (USTC) in 2018.

PUBLICATIONS

1. **ARISE: An Accuracy-Aware Proactive Framework for Serving Concurrent Edge-Assisted AR Clients**
   Z. Jonny Kong*, **Qiang Xu***, and Y. Charlie Hu (* co-primary)
   The 22nd ACM International Conference on Mobile Systems, Applications, and Services (**MobiSys 2024**)

2. **Can 5G mmWave Enable Edge-Assisted Real-Time Object Detection for Augmented Reality?**
   Moinak Ghoshal*, Z. Jonny Kong*, **Qiang Xu***, Zixiao Lu, Shivang Aggarwal, Imran Khan, Jiayi Meng, Yuanjie Li, Y. Charlie Hu, and Dimitrios Koutsonikolas (* co-primary)
   31st International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (**MASCOTS 2023**)

3. **AccuMO: Accuracy-Centric Multitask Offloading in Edge-Assisted Mobile Augmented Reality**
   Z. Jonny Kong*, **Qiang Xu***, Jiayi Meng, and Y. Charlie Hu (* co-primary)
   The 29th Annual International Conference on Mobile Computing and Networking (**MobiCom 2023**)

4. **An In-Depth Study of Uplink Performance of 5G mmWave Networks**
   Moinak Ghoshal*, Z. Jonny Kong*, **Qiang Xu***, Zixiao Lu, Shivang Aggarwal, Imran Khan, Yuanjie Li, Y. Charlie Hu, and Dimitrios Koutsonikolas (* co-primary)
   The 2nd ACM SIGCOMM Workshop on 5G and Beyond Network Measurements, Modeling, and Use Case
   (**5G-MeMU 2022**)

5. **Can 5G mmWave Support Multi-user AR Apps?**
   Moinak Ghoshal, Pranab Dash, Z. Jonny Kong, **Qiang Xu**, Y. Charlie Hu, Dimitrios Koutsonikolas, and Yuanjie Li
   Passive and Active Measurement Conference 2022 (**PAM 2022**)

6. **An Empirical Study on the Impact of Deep Parameters on Mobile App Energy Usage**
   Qiang Xu, James C. Davis, Y. Charlie Hu, and Abhilash Jindal
   The 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (**SANER 2022**)

7. **Do Larger (More Accurate) Deep Neural Network Models Help in Edge-assisted Augmented Reality?**
   Jiayi Meng, Z. Jonny Kong, **Qiang Xu**, and Y. Charlie Hu
   ACM SIGCOMM 2021 Workshop on Network-Application Integration (**NAI 2021**)

8. **Proactive Energy-Aware Adaptive Video Streaming on Mobile Devices**
   Jiayi Meng, **Qiang Xu**, and Y. Charlie Hu
   2021 USENIX Annual Technical Conference (**USENIX ATC 2021**)