UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Augmented Reality on the Network Edge

Permalink
https://escholarship.org/uc/item/82n0w5wj

Author
Ran, Xukan

Publication Date
2021

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Augmented Reality on the Network Edge

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xukan Ran

September 2021

Dissertation Committee:

Dr. Jiasi Chen, Chairperson
Dr. K. K. Ramakrishnan
Dr. Srikanth Krishnamurthy
Dr. Zhijia Zhao

Copyright by
Xukan Ran
2021

The Dissertation of Xukan Ran is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank my advisor, Dr. Jiasi Chen, for her continuous support and guidance. Without her help, this work would not exist. I would also like to thank my dissertation committee, Dr. K.K. Ramakrishnan, Dr. Srikanth Krishnamurthy, and Dr. Zhijia Zhao for their insightful feedback and their support along the way.

I want to thank all my colleagues at WCH 367. Special thanks to Haoliang Chen, Kittipat Apicharttrisorn, Carter Slocum, and Yi-Zhen Tsai for their support during my PhD journey.

This work is based on the following publications:

1, Xukan Ran, Haoliang Chen, Zhenming Liu, Jiasi Chen, “Delivering deep learning to mobile devices via offloading”, ACM Sigcomm VR/AR Network 2017;

2, Xukan Ran, Haoliang Chen, Xiaodan Zhu, Zhenming Liu, Jiasi Chen, “Deepdecision: A mobile deep learning framework for edge video analytics”, IEEE INFOCOM 2018;

3, Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, Amit K Roy-Chowdhury, “Frugal following: Power thrifty object detection and tracking for mobile augmented reality”, ACM Sensys 2019;

4, Xukan Ran, Carter Slocum, Maria Gorlatova, Jiasi Chen, “ShareAR: Communication-efficient multi-user mobile augmented reality”, ACM HotNets 2019;

5, Xukan Ran, Carter Slocum, Yi-Zhen Tsai, Kittipat Apicharttrisorn, Maria Gorlatova, Jiasi Chen, “Multi-user augmented reality with communication efficient and spatially consistent virtual objects”, ACM CoNext 2020.

To my wife, Sha, for everything.

ABSTRACT OF THE DISSERTATION

Augmented Reality on the Network Edge

by

Xukan Ran

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2021
Dr. Jiasi Chen, Chairperson

Mobile Augmented Reality (AR) is becoming more and more popular, with the AR market estimated to grow to $61 billion by 2023. However, there is a lack of understanding of AR performance in terms of accuracy, latency, among others. For example, a virtual object augmented on the table may drift in the air if the AR accuracy is low. The initialization latency for the multi-user AR can be long and will significantly impact user experience. This thesis explores and improves the performance for both deep learning based and SLAM based AR on mobile devices on the network edge.

First, we propose DeepDecision, a deep learning framework that ties together front-end devices with more powerful backend “helpers” (e.g., home servers) to allow deep learning to be executed locally or remotely in the cloud/edge server. The complex interaction between model accuracy, video quality, battery constraints, network data usage, and network conditions is considered to determine an optimal offloading strategy. Our results show DeepDecision achieves better accuracy comparing with the baseline method.

Second, we developed a lightweight change detector which triggers deep neural networks(DNN) execution when there are significant changes in the input video. When there are no significant changes, DNN will not be triggered and a lightweight tracking algorithm will be applied to maintain previous DNN results. The change detector has high accuracy and very low latency. It helps DNN system to save energy and reach real-time processing without offloading.

Third, we propose SPAR, a SPatially consistent AR framework for SLAM-based multi-user augmented reality. SPAR communicates efficiently by only sending the most relevant environment data while maintaining or even improving the accuracy. We also propose a geo-distance filter so that after the virtual object is initially resolved, SPAR can continue to optimize its accuracy. SPAR also has a preliminary automated tool to measure accuracy in both single-user and multi-user cases. Our results show that SPAR has better accuracy and initialization latency comparing with the baseline method.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Thanks to its augmented interactive experience between real world and virtual objects on the screen, mobile Augmented Reality (AR) is becoming increasingly popular these days, with the AR market estimated to grow to $61 billion by 2023 [38]. Many companies are developing mobile AR platforms and integrating AR into their products, such as Apple ARKit[13], Google ARCore[32], and IKEA Place[49]. Meanwhile, mobile AR has spread out to a wide range of fields including gaming(Pokemon Go Buddy Adventure), online shopping(JINS Eyeglasses), education(Apollo's Moon Shot AR), online chatting(Snap Filters) and physical therapy(Complete Anatomy)[88]. In AR, virtual objects are rendered on the display and overlaid on top of a user's field of view (FoV). To provide a seamless integration with the real world, AR app needs to have an understanding of the surrounding real-world environment [43]; for example, The IKEA augmented reality app IKEA Place will place a virtual sofa on a real floor, rather than drawing the sofa unrealistically floating in the air. The AR algorithms behind the scene require the understanding of the environment and can be categorized into two major types: (1) deep learning based AR (*e.g.*, Snapchat Lenses [34]) in which deep learning is used to classify objects in the real world and overlay on top of them; (2) SLAM based AR (*e.g.*, Just A Line [37]) in which visual and inertial sensors are used to create a 3D map of the real world. However, due to the limited resources from mobile devices, only a few AR apps use deep learning which remains as the bottleneck for AR[95]. Although SLAM based AR started to emerge on mobile devices, there is a lack of understanding of the communication requirements and challenges of multi-user AR scenarios where users can share virtual objects associated with the real world.

We build three systems, DeepDecision, change detector and SPAR, to solve the above problem. DeepDecision is a measurement-driven mathematical framework that effectively schedules the deep learning to be executed either locally on the mobile devices or offloaded to the edge server with a certain input video quality based on accuracy and latency targets, network conditions and battery conditions. The change detector is a lightweight machine learning algorithm that detects major changes in the input video. DNN will only be triggered when major changes are detected to save energy and decrease the latency of the DNN system without offloading. SPAR is a multi-user augmented reality application that applies efficient communication strategies to trade off communication latency for spatial consistency of the virtual objects.

Our experiment results suggest that it's possible to balance between accuracy and latency for both deep learning and SLAM based AR. Augmented reality researchers and developers can apply different strategies to fulfill different demands in various scenarios.

1.1 DeepDecision as a solution for deep learning based AR

Deep learning shows great promise in providing more intelligence to augmented reality devices, but few mobile AR apps execute deep learning in real time due to the lack of infrastructure support. Deep learning algorithms are computationally intensive, and mobile devices can't deliver sufficient compute power for real-time processing. For example, Tensorflow's Inception deep learning model can process about one video frame per second on a typical Android phone, preventing real-time analysis[20]. Even with mobile GPU speedup, the typical processing time can still be up to 600 ms(1.7 frames per second), far from real-time processing(30 frames per second)[47]. However, we observe that although mobile devices have limited computation power, sending deep learning to "backend" computers can result in an effective design as long as the tradeoffs between accuracy and latency are well studied.

We have done extensive measurements to understand the tradeoffs between video quality, network conditions, battery consumption and processing delay as well as accuracy. Based on these measurement and tradeoff studies, we develop DeepDecision, a mobile deep learning framework that achieves target accuracy and latency under bandwidth and energy constraints.

Our results show that there exist various tradeoffs between video resolution, model size, where to run the model as well as accuracy and latency. We also observe that latency has a reverse impact on accuracy for real-time deep learning detection, as the results will get stale and thus cause a worse accuracy. We compared our results with remote-only, local-only and a "slim" version of MCDNN[40] and find DeepDecision is able to provide higher accuracy under variable network conditions.

1.2 Change detector as a solution for deep learning based AR without offloading

Although DeepDecision improves the performance of deep learning system, the processing purely on mobile devices without offloading is still slow and energy-consuming. We developed a lightweight machine learning algorithm called change detector to detect changes in the input video. The deep learning will be applied for the first frame but will not be triggered again unless major changes have been detected. When there is no major change, a lightweight tracking algorithm(Lucas-Kanade method[65]) will be applied to maintain DNN outputs. Such scheme helps the DNN system to save energy and reach real-time performance without the help of an edge server.

Our experiments show the change detector has high precision and recall. Utilizing the change detector helps DNN remain or even improve the accuracy, save energy and reach real-time processing.

1.3 SPAR as a solution for SLAM based multi-user AR

Multi-user augmented reality applications thrive on the mobile app store these days. Such applications with a common set of virtual objects viewed by multiple users not only require cooperation among players but also provide interactions between virtual world and the real world[58]. For example, users can share virtual pokemons with each other in Pokemon go buddy adventure at the same physical location; Microsoft Minecraft Earth AR allows players to build virtual structures or earthworks together on the same real-world structure and Google Just A Line allows multiple users to draw virtual graffiti in the same physical space.

Multi-user AR requires network communications in order to coordinate the positions of the virtual objects on each user's display, yet there is currently little understanding of how such apps communicate and how well the application performs. The current multi-user AR applications such as ARCore and ARKit are closed-sourced and there is also a lack of study on how to measure accuracy of AR and multi-user AR applications[32, 13].

We develop SPAR, a multi-user augmented reality application on Android that shares environment and virtual object information among users under WiFi connections. SPAR has two strategies: SPAR-LARGE and SPAR-SMALL, where SPAR-LARGE shares environment information near the virtual objects while SPAR-SMALL only shares environment information scanned at the time when the virtual object is created. The baseline approach is to share all the environment information. To measure the performance of SPAR, we propose an automatic tool to quantify how much the virtual objects' positions inadvertently change in time and space.

We measure the performance of SPAR in three different mobility patterns and conclude that both SPAR-LARGE and SPAR-SMALL have better accuracy in all the scenarios while SPAR-SMALL has poor latency performance in one scenario due to limited shared environment information.

Chapter 2

DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics

2.1 Introduction

Deep learning shows great promise to provide more intelligent video analytics to augmented reality (AR) devices. For example, real-time object recognition tools could help users in shopping malls [26], facilitate rendering of animations in AR apps (*e.g.*, detect a table, and overlay a game of Minecraft on top of it), assist visually impaired people with navigation [50], or perform facial recognition for authentication [90].

Today, however, only a few AR apps use deep learning due to insufficient infrastructure support. Object recognition algorithms such as deep learning are the bottleneck for AR [95] since they are computationally intensive, and the front-end devices are often ill-equipped to execute them with acceptable latencies for the end user. For example, Tensorflow's Inception deep learning model can process about one video frame per second on a typical Android phone, preventing real-time analysis [20]. Even with speedup from the mobile GPU [48, 47], typical processing times are approximately 600 ms, which is less than 1.7 frames per second. In industry, while a few applications run deep learning locally on a phone (*e.g.*, Apple Photo Album), these are lightweight models that do not run in real time. Voice-based intelligent personal assistants (*e.g.*, Alexa, Cortana, Google Assistant, and Siri) mostly transfer the input data to more powerful backends and execute the deep learning algorithms there. Such cloud-based solutions are only applicable when network access is reliable.

Our observations. While front-end devices are computationally weak, and sending deep learning jobs to "backend" computers is inevitable, the following new observations will yield an effective design:

Tradeoffs between accuracy and latency. AR apps relying on deep learning have different accuracy/latency requirements. For example, AR used in shopping malls for recommending products may tolerate longer latencies (fine to let users to wait a second or two) but have a higher accuracy requirement. In an authentication system that uses deep learning, users could wait even longer but expect ultra-high accuracy.

Sources of latency. When a deep learning task needs to be executed remotely, both the data transmission time over the network and the deep learning computation time can introduce latencies. Prior works (*e.g.*, [47, 40]) focus on optimizing *computation latencies* (*i.e.,* time between the job's arrival at the computation node and the job's completion) by designing sophisticated local scheduling algorithms. We observe that network latencies are often much longer than the computation latencies, so it is important to optimize the offloading decision along with the local processing. Furthermore, although deep learning based real-time video analytics are known to be computationally intense, simple consumer-grade GPUs suffice for most real-time video analysis (*e.g.*, object detections). Thus, home computers could be used as "backend helpers" that are dedicated to a small group of users like family members. In other scenarios, home desktops may not even be needed, as wearable devices such as smartwatches or head-mounted displays could send computation to a user's smartphone nearby.

Video streams and deep learning models as "first class citizens." Deep learning in an AR setting is primarily responsible for interpreting data collected from a camera (*i.e.,* video data). Prior works (*e.g.*, [20, 47, 40]) treat the videos merely as sequences of images and the deep learning models as rigid computation devices that produce uniform forecasting quality. Yet these assumptions lead to the illusion that we face a canonical scheduling problem: a fixed set of computation tasks needs solving, and each deep learning model consumes a predictable amount of resources and produces predictable output (*i.e.,* forecasting quality is known). The assumption, however, will substantially reduce the system performance because they ignore the compressibility of both deep learning models and video streams.

Instead, we ought to treat video streams and deep learning modules as "first-class citizens" and directly optimize the tradeoffs between video and prediction qualities. Specifically, video data should not be treated as a sequence of images (*i.e.,* independent computation tasks) because this will over-consume network bandwidth; instead, aggressive leverage of existing technologies for compressing videos (including DFT, delta coding, and changing resolution *etc.*) will result in the best use of network bandwidth. Certainly, over-aggressive compression may cause declines in video analysis quality. Our solution aims to find the most suitable video encoding scheme that gives the optimal tradeoff between network consumption and prediction quality.

Our contribution. We propose a distributed infrastructure, DeepDecision, that ties together computationally weak front-end devices (assumed to be smartphones in this work) with more powerful back-end helpers to allow deep learning to choose local or remote execution. The back-end helpers can be any devices that supply the requisite computation power. Our solution intelligently uses current estimates of network conditions, in conjunction with the application's requirements and specific tradeoffs of deep learning models, to determine an optimal offload strategy. In particular, we focus on executing a convolutional neural network (CNN) designed for detecting objects in real-time for AR applications. (A similar framework could be applied to any application that requires real-time video analytics.) We seek to understand how the changes of key resources (*e.g.*, network bandwidth, neural network model size, video resolutions, battery usage) in the system impact the decision of where to compute. An overview of our system is illustrated in Fig. 2.1 and Fig.2.2.

We make the following contributions:

1. Extensive measurements of deep learning models on smartphones to understand the tradeoffs between video compression, network conditions and data usage, battery consumption, processing delay, frame rate, and machine learning accuracy;

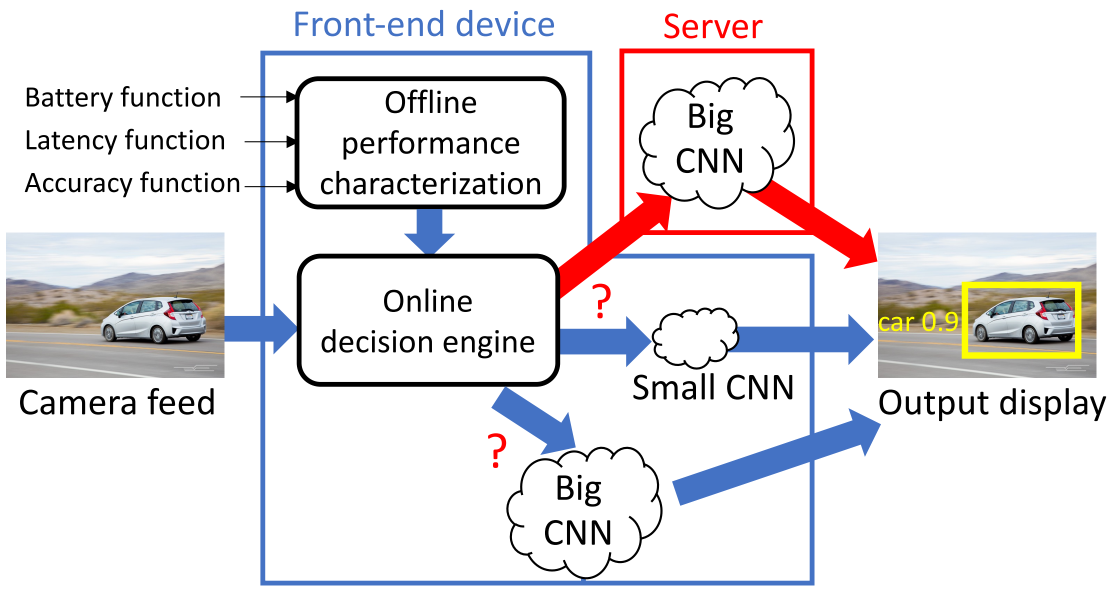


Figure 2.1: System overview. The front-end device chooses where to analyze the input video for real-time display.

2. A measurement-driven mathematical framework that efficiently solves an optimization problem, based on our understanding of how the above parameters influence each other;

3. An Android implementation that performs real-time object detection for AR applications, with experiments that confirm the superiority of our approach compared to several baselines.

Organization. Sec. 2.2 explains the background on neural networks, Sec. 2.3 describes our model and algorithm, and Sec. 2.4 shows our measurements and experimental results. Finally, Sec. 2.5 discusses related work and Sec. 2.6 summarizes.

2.2 Background, Metrics, and Degrees of Freedom

Background. We first provide relevant background on CNNs for video analytics and AR. In video analytics, object recognition (classifying the object) and object detection (locating the object in the frame) are both needed. In AR, the processing pipeline also includes drawing an overlay on top of the located and classified object. Neural nets are the state-of-the-art in computer vision for object recognition and detection, and many existing neural nets for object recognition build a pipelined solution, *i.e.,* they use one neural net to detect the boundaries of objects and a second net to inspect contents within each bounding box. In this work, we use a particular CNN called Yolo [82]. (Our framework can also be adapted to other popular CNNS such as [83, 63].) Yolo is optimized for processing video streams in real-time and possesses two salient features: 1. *One neural net for boundary detection and object recognition.* Observing that using multiple neural nets unnecessarily consumes more resources, Yolo trains one single neural network that predicts boundaries and recognizes objects simultaneously. 2. *Scaling with resolution.* Yolo handles images with different resolutions, *e.g.*, when there is a change in the dimension of an input to a convolutional layer, Yolo does not change the kernels and their associated learnable parameters – this would result in a change of output dimensions. Thus, the compute time of Yolo scales directly with input's resolution, *e.g.*, lower resolution images require less computation.

Key performance metrics. AR apps often require service guarantees on two important metrics:

1. Frame rate: The frame rate is the number of frames that we feed the deep learning model per second when it's running locally, or is the video frame rate (FPS) when offloading.

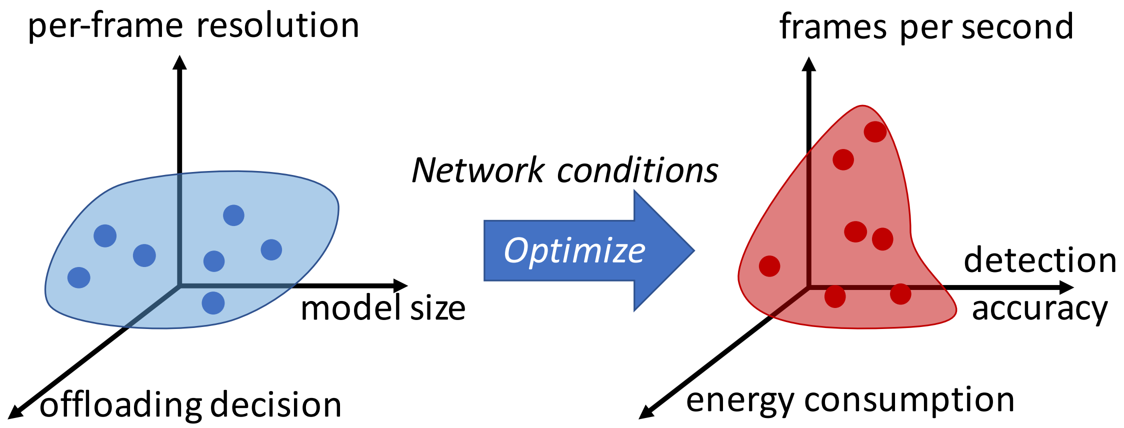


Figure 2.2: Input parameters and outcomes.

2. Accuracy: The accuracy is a metric that captures (a) whether the object is classified correctly; (b) whether the location of the object in the frame is correct.

Being a responsible citizen. While purely focusing on these key metrics may maximize the performance of the video analytics module, other potential impacts on the frontend device should be considered. For example, running more powerful deep learning models will consume extra CPU cycles, disrupting other background processes, and draining the device battery. If the client communicates with the server, the network transmission also uses battery; Also, if the data transfer is over LTE, the monetary cost to the user in terms of data quota must also be considered. These factors of battery consumption and network data usage must be considered holistically alongside the key performance metrics.

Degrees of freedom. There are several degrees of freedom we consider in this work as shown in left part of Fig. 2.2.

1. Adjust the frame resolution. By decreasing the frame resolution, we can decrease the execution time of a deep learning model, which also lowers the energy cost. However, this may also decrease the accuracy of the model. Conversely, increasing the frame resolution may increase the accuracy, at the expense of lower frame rate and greater energy drain.

2. Use smaller deep learning models. We may wish to use a smaller neural network to reduce the run time and the energy cost, at the cost of reducing the prediction accuracies. Conversely, using a larger deep learning model increases the run time and energy, but boosts the accuracy.

3. Offload to backend. By sending the computation job to a backend server, we can substantially reduce the computational burden at front-end devices, increasing the frame rate. However, this may result in extra startup delay from the network transfer, causing the server's result to be stale by the time it is returned to the client, thus decreasing the prediction accuracy.

4. Compress the video. When offloading, one may wish to compress the video more/less based on the network conditions. Choosing a low target video bitrate reduces the network transmission time, but potentially decreases the accuracy. Conversely, using a high video bitrate may result in higher accuracy but will also result in longer transmission time, making the detection results stale and decreasing the accuracy.

5. Sample the video at lower frequencies. We do not need to process every frame in the video; instead we may sample only a small fraction of frames for further processing. In this way, we may reduce the total computation demand of an AR app.

Each of these operations may impact one or more of the key performance metrics described above. Furthermore, we may employ multiple operations simultaneously, *e.g.*, we can reduce the resolution and use smaller models at the same time. In fact, any subset of these operations defines a legitimate strategy, although not necessarily optimal.

2.3 Our algorithmic problem

This section describes the problem, our optimization framework, and the algorithms to solve this optimization problem. **Challenges:** (1) As can be seen from the previous section and Fig. 2.2, the interactions between the degrees of freedom and the key performance metrics are complex. For example, some decision variables increase one key metric but decrease another metric (*e.g.*, higher resolution increases accuracy but decreases frame rate); or some decision variables may affect the same metric in multiple ways (*e.g.*, transmitting the video at a higher bitrate could increase the accuracy, but could also increase the latency, which decreases accuracy. See Fig. 2.7). Selecting the right combination of decision variables that maximizes the key metrics, while satisfying energy, cost, and performance constraints is no easy task. (2) Moreover, many of these tradeoffs cannot be expressed cleanly in analytic form, making any solution or analysis difficult. For example, analyzing the relationship between staleness and accuracy is a challenging task that depends on the video content, the particular deep learning model being used, the resolution of the video, and the compression of the video. The lack of analytic understanding of these tradeoffs is in part due to the complexity of the deep learning models themselves, whose theoretical properties are not yet well understood.

Our approach: Our approach is therefore to create a *data-driven* optimization framework that takes as input the empirical measurements of these tradeoffs, computes

Variable	Description
p	frame resolution (pixels^2)
r	video bitrate (bits/s)
f	frame rate (frames/s)
$y_i(t)$	model decision at time t
$a_i(p, r, \ell_i)$	accuracy of model i (%)
$b_i(p, r, f)$	battery of model i (J/s if $i = 0$, J/frame if $i > 0$)
$\ell_i(p, r, f)$	total delay when using model i (s/frame)
$\ell_i^{\text{CNN}}(p)$	processing delay from running model i
B	network bandwidth (kbps)
L	network latency (s)
$\mathcal{B}$	target battery usage (J/s)
A	accuracy target (%)
F	frame rate target (frames/s)
c	monetary cost (\$/bit, if use cellular network)
C	target monetary cost (\$/bit)
α	parameter that trades off accuracy for frame rate in the objective function

Table 2.1: Table of Notation. $i = 0$ represents remote execution, and $i = 1, \dots, N$ represents local execution on the front-end device.

the optimal combination of decision variables that maximizes the key metrics, and outputs the optimal decision. Our framework must be general enough to handle any values of input measurement data while still capturing the tradeoffs between decision variables and metrics. In this section, we will describe the optimization framework that we designed and its solution; while Sec. 2.4, we show the actual input data based on our measurements with real systems, as well as the experimental results from our Android application.

In the DeepDecision system, we divide time into windows of equal size, and solve an optimization problem at the beginning of each interval to decide the deep learning algorithm's configurations: the frame rate sampled by the camera (f), the frame resolution (p), the video bitrate (r), and which model variant to use (y_i). The problem is:

Problem 1

$$\underset{p,r,f,\mathbf{y}}{\textit{maximize}} \quad f + \alpha \left(\sum_{i=0}^{N} a_i(p, r, \ell_i) y_i \right) \tag{2.1}$$

$$\textit{s.t.} \quad \ell_i = \begin{cases} \ell_i^{\text{CNN}}(p) + \frac{r}{f \cdot B} + L & \textit{if } i = 0 \\ \ell_i^{\text{CNN}}(p) & \textit{if } i > 0 \end{cases} \tag{2.2}$$

$$\sum_{i=0}^{N} \ell_i^{\text{CNN}}(p) y_i \leq 1/f \tag{2.3}$$

$$r \cdot y_0 \leq B \tag{2.4}$$

$$\sum_{i=0}^{N} b_i(p, r, f) y_i \leq \mathcal{B} \tag{2.5}$$

$$c \cdot r \cdot y_0 \leq C \tag{2.6}$$

$$f \geq F \tag{2.7}$$

$$\forall i : a_i(p, r, f) \geq A \cdot y_i \tag{2.8}$$

$$\sum_{i=0}^{N} y_i = 1 \tag{2.9}$$

$$\textit{vars} \quad p, r, f \geq 0, y_i \in \{0, 1\} \tag{2.10}$$

The objective (2.1) is to maximize the number of frames sampled plus the accuracy of each frame (see Key Performance Metrics in Sec. 2.2). The relative importance of accuracy versus frame rate is determined by parameter α. Constraint (2.2) says that the total delay experienced by a frame is equal to the CNN's processing time plus the network transmission time (if applicable). Constraint (2.3) says that the frame rate cannot be chosen to exceed the processing time of the CNN (remote or local). Constraint (2.4) says that the video cannot be uploaded at a higher bitrate than the available bandwidth. Constraint (2.5) says that the battery usage cannot exceed the maximum target. Constraint (2.6) says that the monetary cost cannot exceed the maximum target. Constraints (2.7) and (2.8) allow the application to define a minimum required frame rate and accuracy. Constraint (2.9) says that only one model may be selected (remote or one of the local models).

If the frame rate, resolution, and video bitrate were known, then Prob. 1 is a multiple-choice multiple-constraint knapsack program, where the items are the model variants, an item's utility is the model's accuracy, and an item's weight is in terms of latency, bandwidth, battery, and monetary cost. The multiple-choice comes from the fact that for each frame, there is a choice to offload, or process locally (choosing a model size). The multiple-constraint comes from the latency, bandwidth, battery, and cost constraints. However, the key difference from the classical problem is that the utility and costs of the items are also functions of the optimization variables. Moreover, they are generally non-linear functions, and must be determined empirically from measurements.

A brute-force solution to Prob. 1 would take $O(r_{\max} \cdot f_{\max} \cdot p_{\max} \cdot N)$. The brute-force solution is impractical when we need to frequently make new decisions and/or carry out a grid-search in fine granularity. We need to leverage the mathematical structure of the problem (based on simple intuitions and confirmed by extensive measurements in Sec. 2.4) to design a more efficient algorithm. These intuitions are:

1. Accuracy: For remote models, the accuracy per frame depends on a number of factors: the video resolution, the video bitrate/compression, and the end-to-end delay (which is itself a function of resolution, bitrate, frame rate, and network latency and bandwidth), i.e. $a_0(p, r, \ell_0) = a_0(p, r, \ell_0(p, r, f, B, L))$. For local models, the accuracy model can be simplified because it depends only on the resolution and delay, since we do not need to compress the video for transmission over the network, *i.e.,* $a_i(p, r, \ell_i) = a_i(p, \ell_i), i > 0$.

2. Battery: When transmitting to the server, the energy per time depends only on how much data is transmitted over the network, *i.e.,* $b_0(p, r, f) = b_0(r, B)$. For local models, the battery usage per time depends on the resolution and the number of frames processed, *i.e.,* $b_i(p, r, f) = f \cdot b_i(p), i > 0$.

3. Latency: The latency per frame when running the CNN locally depends only on the resolution, since a larger resolution requires more convolutions to be performed, *i.e.,* $\ell_i(p, r, f) = \ell_i^{\text{CNN}}(p), i > 0$. When offloading to the server, however, the total latency is a function of the CNN processing time plus the network transmission time, *i.e.,* $\ell_0(p, r, f) = \ell_0^{\text{CNN}}(p) + \frac{r}{f \cdot B} + L$.

Algorithm 1 DeepDecision algorithm

Input: Target cost C, target battery $\mathcal{B}$, cost per bit c, network bandwidth B, network latency L, model battery usage function b_i, model latency function ℓ_i, model accuracy function a_i

Output: Frame resolution p^*, video bitrate r^*, frame rate f^*, decision of model variant $\mathbf{y}^*$

```
 1: $f \leftarrow \frac{1}{f_0^{\text{CNN}}}$                                   ▷ remote model
 2: $p, r \leftarrow \arg\max_{p, r \le \min(B, b_0^{-1}(\mathcal{B}|B), \frac{C}{c})} (f + a_0(p, r, \ell_0^{\text{CNN}} + \frac{r}{fB} + L))$
 3: if $a_0(p, r, f) \ge A$ and $f \ge F$ then
 4:     $u_{\max} \leftarrow f^* a_0(p, r, f)$
 5:     $r^* \leftarrow r, f^* \leftarrow f, p^* \leftarrow p, \mathbf{y}^* \leftarrow e_i$
 6: for $i \leftarrow 1$ to $N$ do                               ▷ try the local models
 7:     $r \leftarrow r_{\max}$                                  ▷ don't need to compress locally
 8:     $p \leftarrow \arg\max_p(\min(\frac{1}{\ell(p)}, \frac{\mathcal{B}}{b(p)}) + a_i(p, \ell_i(p))$
 9:     $f \leftarrow \min(\frac{1}{\ell(p)}, \frac{\mathcal{B}}{b(p)})$
10:     $u \leftarrow f + a_i(p, \ell_i(p))$
11:     if $u > u_{\max}$ and $a_i(p) \ge A$ and $f \ge F$ then
12:         $u_{\max} \leftarrow u$
13:         $r^* \leftarrow r, f^* \leftarrow f, p^* \leftarrow p, \mathbf{y}^* \leftarrow e_i$
14: return $p^*, r^*, f^*, \mathbf{y}^*$
```

We are now ready to describe our Alg. 1 that exactly solves Prob. 1 and improves on the brute-force search efficiency. With some abuse of notation, we define the inverse of a function $g : \mathbb{R}^2 \to \mathbb{R}$ as $g^{-1}(y|z) = \arg\max_x (g(x,z) : g(x,z) \leq y)$.

- Line 1-5, $i = 0$ (remote model): The constraints are: $\ell_0 = \ell_0^{\text{CNN}} + \frac{r}{f \cdot B}, \ell_0^{\text{CNN}} \leq \frac{1}{f}, b_0(r, B) \leq \mathcal{B}, c \cdot r \leq C$, and $r \leq B$. Since the objective increases with frame rate (the f term grows and $\frac{r}{f \cdot B}$ shrinks, resulting in higher accuracy), we can pick the maximum frame rate that satisfies the constraints. We next search across resolution p and bitrate r to find the best combination. Lastly, we check if the frame rate and accuracy constraints are satisfied.

- Line 6-13, $i > 0$ (local models): The constraints are: $\ell_i(p) \leq \frac{1}{f}$ and $f \cdot b_i(p) \leq \mathcal{B}$. Since the processing is local, the video bitrate is set to the maximum. The tradeoff is between the resolution p and frame rate f (setting a higher resolution improves accuracy, but the CNN takes longer, decreasing the frame rate). For each value of p, since $b_i(p)$ and $l_i(p)$ are non-decreasing, we can find the maximum f that still satisfies the constraints, then pick the best p overall. Finally, we check if the frame rate and accuracy constraints are satisfied.[1]

The battery, latency, and accuracy functions $b_i^{-1}, \ell_i^{-1}, a_i$ can easily be pre-stored (by using hash tables) so that their lookups take constant time. We use linear interpolation on these functions if the measurement density is insufficient. The running time of our

[1]We remark that when the frame rate does not satisfy (2.7), we could not have picked a different frame rate resulting in a feasible solution. This is because the algorithm picks the maximum possible f for each value of p. A similar argument also holds for accuracy constraint A.

algorithm takes $O(p_{max}(r_{max} + N))$, which is a significant improvement over the brute-force solution.

We next make a number of remarks. **1. Utility function.** The utility function in (1) is the sum of frame rate and accuracy. One may also consider other utility functions, *e.g.*, multiplicative $f \cdot \sum_{i=0}^{N} a_i(p, r, \ell_i)y_i$, where the intuition is that utility depends on collecting more frames each with high accuracy. Alg. 1 is still applicable when the utility function changes (so long as it is monotone in both frame rate and accuracy). We have experimented with the multiplicative objective function and found that it tends to emphasize frame rate at the expense of accuracy, and choose solutions with low accuracy but high frame rate. Therefore, we use the additive objective function (2.1) in the remainder of this work.

2. Budgets in each time interval. In the current formulation, the user inputs her battery and monetary constraints as an average usage over time (*e.g.*, $\$/s, J/s$). Alternatively, the user may wish to specify total battery and monetary budgets in each time period (*e.g.*, $\$, J$), and have the algorithm use dynamic programming to make online decisions. But the multi-stage optimization approach is unlikely to be effective in our setting, as such an optimization often requires knowledge of the distribution of the users' future actions. Predicting users' future actions is remarkably difficult, and is an area we intend to explore in the future based on prior literature in similar domains [71].

3. Time dynamics. Alg. 1 runs periodically, and re-computes a new solution based on current network conditions. For example, if a local model is currently being used, and the network bandwidth improves, DeepDecision may decide to change to offloading.

However, if the network bandwidth will only increase temporarily, it may be suboptimal to switch due to cost overhead (*e.g.*, loading a new deep learning model or establishing a new network connection takes time). To reduce frequent oscillations, we analyze the conditions under which a switch should occur, and add this as an outer loop to Alg. 1. We assume there is a throughput predictor that can estimate the future bandwidth and latency over a short period of time [93]. Let T be the length of time of the network conditions change, f_l^*, a_l^* be the optimal solution of Prob. 1 assuming the model decision is fixed to be local, and f_r^*, a_r^*, r_r^* be the optimal solution of Prob. 1 assuming the model decision is fixed to be remote. Due to space constraints, the proof can be found in Sec. 2.7, and here we present the results directly. DeepDecision should switch from local to remote iff:

$$f_r^* - f_l^* \leq \alpha(a_l^* - a_r^*) \tag{2.11}$$

And switch from local to remote iff:

$$\alpha(a_l^* - a_r^*) \leq f_r^* \left(1 - \frac{1}{T}\left(\frac{r_r^* f_r^*}{B} + L\right)\right) - f_l^* \tag{2.12}$$

Note that the conditions are asymmetric because switching from local to remote incurs additional delay while waiting for the first result to arrive from the server, thus decreasing the average frame rate in the objective function (2.1). Intuitively, the factors that encourage switching from local to remote are: long period of time T of improved bandwidth, high bandwidth B, and low network latency L.

2.4 Measurements & Experiments

This section describes our experiments, which serve two purposes. First, we want to understand the interactions between various factors (*e.g.*, processing time, video quality, energy consumption, network condition, the accuracies of different deep learning models) on both the local device and the server. While prior works have carried out limited profiling of running deep learning on servers [46] or on phones [40], to the best of our knowledge, we are the first to explicitly consider the input stream as a video rather than a sequence of images, as well as the impact of network conditions on the offloading strategy, and the tradeoffs of compressible deep learning models. Second, we seek to understand our algorithm's behavior compared to existing algorithms and assess its ability to make decisions on where to perform computation. These baseline comparison algorithms include:

1. Remote-only solution: All frames are offloaded to the backend. Many industrial solutions (*e.g.*, Alexa, Cortana, Google Assistant, Siri, *etc.*) adopt this solution.

2. Local-only solution: All jobs are executed locally. Some specific applications, such as Google Translate, run a compressed deep learning model locally.

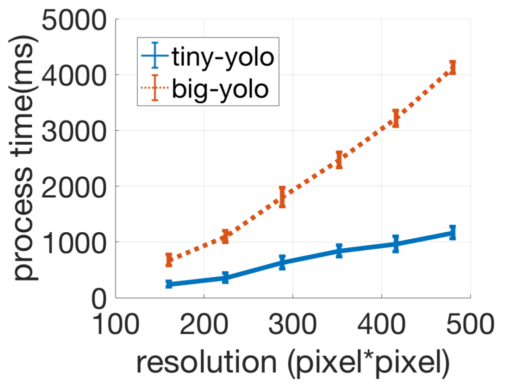
3. Strawman: We implement a "slim" version of MCDNN [40] optimized for the scenario in which the device serves one application. Our strawman picks the model variant with the highest accuracy (defined below) that satisfies the remaining monetary or energy budget. We note that MCDNN does not consider the effects of network bandwidth/latency, how often the neural net should execute, or the impact of delay on accuracy.

2.4.1 Testbed Setup

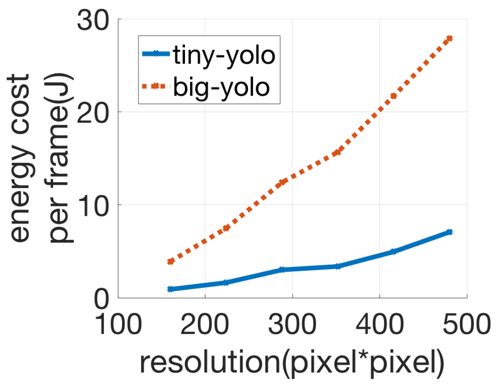
Our backend server is equipped with a quad-core Intel processor running 2.7 GHz with 8 GB of RAM and an NVIDIA GeForce GTX970 graphics card with 4GB of RAM. Our front-end device is a Samsung Galaxy S7 smartphone.[2] We develop an Android version of Yolo based on Android Tensorflow [4] and the Darknet deep learning framework [82]. The Android implementation can run a small deep learning model (called *tiny-yolo*) with 9 convolutional layers, and a bigger deep learning model (called *big-yolo*) with 22 convolutional layers. Both models can detect 20 object classes and are trained on the VOC image dataset [27]. The server runs *big-yolo* only.

When offloading is chosen by Alg. 1, the front-end device compresses the video frame and chooses the correct frame rate and resolution, then sends the video stream to the server. The stream is sent using RTP running on top of UDP [3]. Videos are compressed using the H.264 codec at one of three target bitrates (100kbps, 500kbps, and 1000kbps). The video frame rate can be set between 2 and 30 frames per second (FPS), and the video resolution can be set to 176×144, 320×240, or 352×288 pixels. Our app also logs the battery usage reported by the Android OS, the data usage, and the time elapsed between sending the frame and receiving the detection result from the server. We feed a standard video dataset [5] to the smartphone to ensure a consistent testing environment for the different algorithms.

[2]We also tested on other smartphones such as the Google Pixel and OnePlus 3T and found similar qualitative behaviors. Our system can work on any front-end device by first running performance characterization offline.



(a) process time vs. resolution



(b) energy vs. resolution

Figure 2.3: Tradeoffs on the front-end device. (2.3a): Processing time increases with resolution, especially for *big-yolo*. (2.3b): Energy usage increases with resolution, especially for *big-yolo*.

Accuracy metric. We measure accuracy using the Intersection over Union (IOU) metric, similar to [20]:

$$\text{IoU} = \frac{area(R \cap P)}{area(R \cup P)} \tag{2.13}$$

where R and P are the bounding boxes of the ground truth and the model under evaluation, respectively. The average of the object IoUs in the frame gives the frame's IoU. The average of the frame IoUs gives the video's IoU. Our ground-truth model is *big-yolo* executed on raw videos (without any compression) at the 352×288 resolution (we select this particular resolution out of convenience since pre-trained models are available).

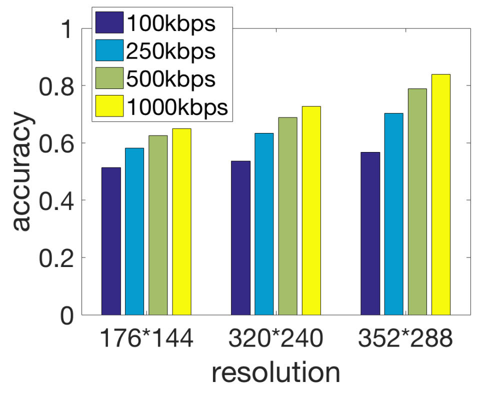
2.4.2 Measuring tradeoffs without network effects

We first study when the phone runs the deep learning locally, without offloading, to understand baseline performance.

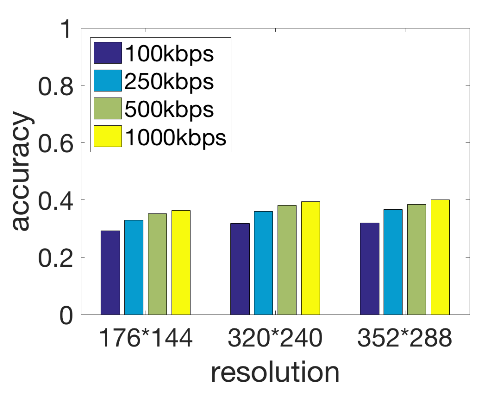
Impact of video resolution: We vary the image resolution from 160×160 to 480×480 pixels. Recall that Yolo can dynamically adjust its internal structure for different resolutions, so its running time is sensitive to the video resolution. In Fig. 2.3a, we plot the tradeoff between frame resolution and processing time. When the CNN runs on the phone, *big-yolo*'s processing time is between 600ms and 4500ms, whereas *tiny-yolo*'s processing time is between 200ms and 1100ms. Since the processing time increases, we also expect the battery usage to increase. In Fig. 2.3b, we show the tradeoff between frame resolution and energy consumption per frame. We note that both processing time and energy consumption scale linearly with the width/height of a video. These two functions are used as input to Alg. 1 (specifically, $\ell_i^{\text{CNN}}(p)$ and $b_i(p)$, for $i > 0$). We also measure the energy of offloading, and find its mean value to be 2900 mW, independent of bitrate and frame rate.

Parameterizing accuracy. We next study the correlation between the accuracies of deep learning models under different image qualities. We use two parameters to determine the clarity of a video/image sequence. *1. Resolution:* This is intuitive, because higher resolution often corresponds to better image quality; and *2. Bitrate:* Resolution by itself does not determine the image quality, as the number of bits used to encode that resolution also matters. A low bitrate will cause the video encoder to aggressively compress the frames and create distortion artifacts, decreasing the video quality and the prediction accuracy.

We seek to understand how the video resolution and bitrate interact with the model accuracy. To do this, we encode the videos in different combinations of resolutions and target bitrates, measure their accuracy, and show the results in Fig. 2.4. Which factor is more important for accuracy, the resolution or the bitrate?



(a) Accuracy of *big-yolo*.



(b) Accuracy of *tiny-yolo*.

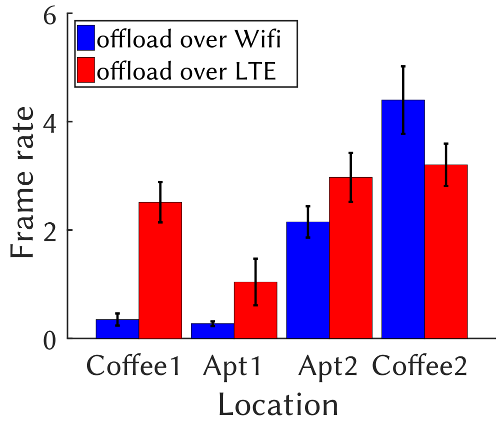
Figure 2.4: Model accuracy for different video qualities. Accuracy increases with resolution and bitrate.

We observe that increasing the resolution without increasing the bitrate has a limited impact on the prediction quality; for example, in Fig. 2.4a the 100kbps bar stays (almost) flat for different resolutions. However, if the bitrate increases along with resolution, there can be substantial accuracy improvement, as shown in Fig. 2.4a for the high-resolution case.

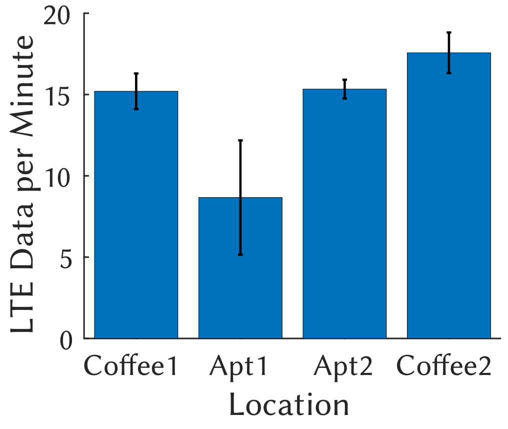
The non-linear interactions between bitrate, resolution, and accuracy suggest the need for a sophisticated offloading decision module that will carefully consider the complex tradeoffs between the various resources. In Alg. 1, these tradeoffs are captured by the function $a_0(p, r, \ell_0)$.

2.4.3 Preliminary study of network effects

In this subsection, we measure the preliminary offloading performance in-the-wild at various locations, including a home, coffee shop, and university campus. We offload images frames to the server and consider several test locations: when the client is in the



(a) Latency



(b) LTE Data usage

Figure 2.5: Performance in-the-wild. Using LTE to increase the frame rate comes at the expense of data quota usage.

same subnet as the server, a different subnet but in the same city, and finally when the client is in a different city than the server. Our eventual goal is to see if the network difference has significant impact on the system and study the possible LTE cost of such application.

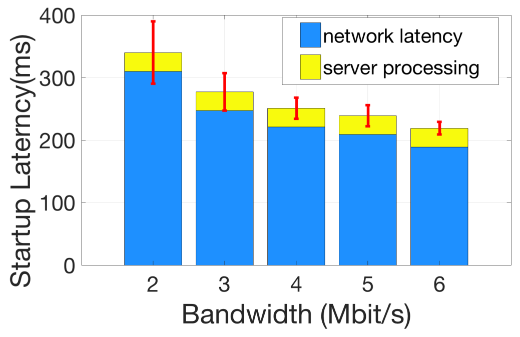
Specifically, our test locations are:

- Coffee 1 (different subnet, different city): A coffee shop in Berkeley, CA. The camera is pointed towards a window and detects cars and people on the street.
- Apt 1 (different subnet, different city): An apartment in Berkeley, CA with cable Internet. The scene is a fairly static home environment and mainly detects computer monitors, cups, and potted plants.
- Apt 2 (different subnet, same city): An apartment in Riverside, CA. The main objects detected are chairs, refrigerators, and TV monitors.
- Coffee 2 (same subnet, same city): An on-campus coffee shop in Riverside, CA. It mainly detects chairs, tables, and umbrellas.

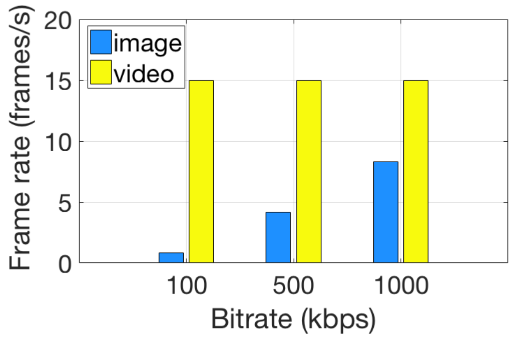
In Fig. 2.5a, we plot the frame rate of the client in these locations, when the client offloads using WiFi or LTE. Each trial lasts 60 seconds, and repeated 3 times. In general, the frame rate with WiFi in a city far from the server (Coffee1 and Apt1) show quite a low frame rate, with each frame taking more than 500 ms to process. When the client is located in the same city as the server (Coffee 2 and Apt 2), the frame rate over WiFi seems to be slightly better, particularly if the client is located in the same subnet as the server. Qualitatively, we observe that in high-latency environments such as the coffee shop, the detection boxes become inaccurate and are drawn in incorrect locations on the screen. The reason is because if the camera or the object is moving and the network is slow, the result returned from the server is stale. We have a more detailed analysis in Sec. 2.4.4 and Fig. 2.7.

In general, the performance (both in terms of quantitative frame rate and qualitative observations) over LTE seems to be better; however, there is a tradeoff here. Although LTE may provide a higher frame rate, the typical constraint is data usage, which is limited and costs money. To delve further into the monetary costs to the user, Fig. 2.5b shows the LTE data usage in the same scenarios. We can see that the average data usage is about 15 MB/minute, which is fairly high. Assuming a 2GB costs \$35, this mean that each minute of usage costs \$0.25. Another possible tradeoff is with accuracy: if LTE has higher bandwidth, the user can upload higher resolution video frames for higher accuracy, at the expense of paying more for more data transfers.

With these results in mind, we perform more experiments to study the network impact and encode the image frames to video streams to save potential LTE cost.



(a) Startup latency vs. bandwidth



(b) Video vs. image offloadings

Figure 2.6: Tradeoffs on the cloud. (2.6a): Network latency dominates compute latency. (2.6b): Compressing the offloaded video enables higher frame rates than image offloading.

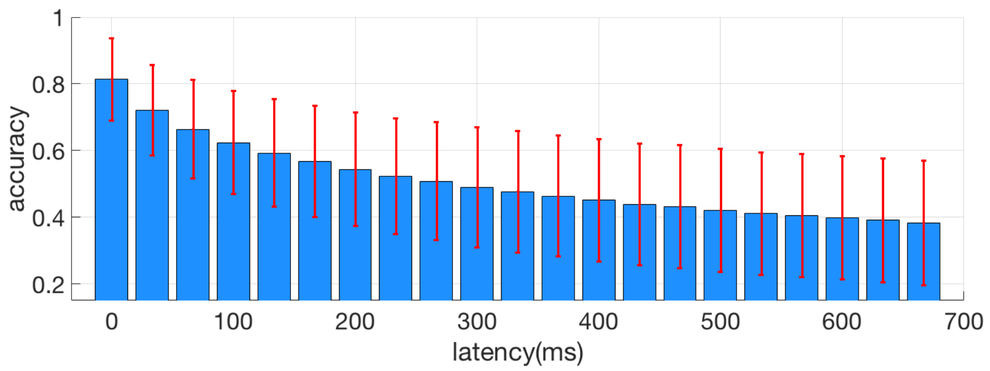


Figure 2.7: Accuracy as a function of total latency (model processing time + network transmission time). Accuracy decreases as the latency increases, due to stale frames, especially for high-motion videos.

2.4.4 Measuring tradeoffs with network effects

Next, we study the impact of the (communication) network conditions on system performance. The Samsung Galaxy S7 phone is located in the same subnet as the server, and for the sake of these measurements, always chooses to offload. We use the `tc` traffic control tool to emulate different network conditions and use data usage to estimate LTE monetary cost.

End-to-end latency. The end-to-end latency l_i experienced by the viewer is the sum of the processing latency ℓ_i^{CNN} plus the network transmission time. We measure the end-to-end latency of each frame as well as the frame rate, when varying the bandwidth and latency between the client and the server. The latency with unconstrained bandwidth between the client and the server is about 30 ms. We repeat each trial 30 times with a frame resolution of 352×288 pixels, and plot the results in Fig. 2.6a. We observe that the network transmission time consumes the majority of the total latency, while the server's *big-yolo* CNN generally executes in less than 30 ms. This indicates that network transmission is the key driver of latency and frame rate, rather than the CNN processing time, and that any offloading strategy should be highly aware of the current network conditions when making a decision.

Impact of latency on accuracy. Network latencies can cause delayed delivery of the output, decreasing the accuracy. For instance, suppose at time $t = 0$, a frame is sent to the back-end server, processed, and the result returned to the front-end device at $t = 200$ ms. At that time, if the scene captured by the camera has changed (for example due to user mobility), the detected object location, and thus the overlay drawn on the display, may be inaccurate. Hence, any system that performs offloading of real-time scenes must take this delay into account when measuring the accuracy; however, previous works generally compute the accuracy relative to the original time of frame capture [40, 47]. To understand the impact of latency on accuracy, for each video at fixed (resolution, bitrate), we measure how the accuracy changes as a function of latency (i.e., how changing the round-trip time will affect prediction accuracy). A sample result for 1000 kbps, 30 FPS videos at 352×288 resolution is shown in Fig. 2.7, where the height of the bar is the mean accuracy across videos and the error bar is the standard deviation. We observe that accuracy decays slowly for these particular videos, which have relatively little motion. One qualitative observation we make is that videos with more active subjects (*e.g.*, foreman [5]) tend to have much shorter half-life than videos with "talking heads" (*e.g.*, akiyo [5]). The relationship between accuracy and latency is modeled as $a_0(p, r, \ell_0)$, where ℓ_0 is the latency.

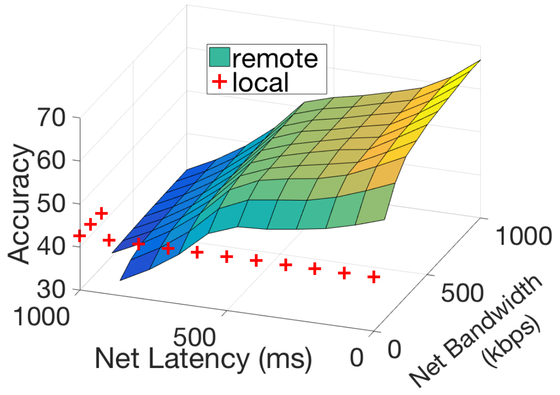
Video compression. Finally, DeepDecision also leverages the benefits of video compression. In contrast to previous works which mainly consider videos as a sequence of images, our system encodes the video as a group-of-pictures with I and P frames, significantly reducing the network bandwidth.

To show this, we measure the frame rate of the offloaded scene when the scene is encoded as an image versus as a video. (Specifically, to compute the image frame rate, we divide the network bandwidth by the size of each frame when saved as an independent image.) We plot the results for different target bitrates in Fig. 2.6b, and observe that encoding the scene as a video can help us send 10× more frames to the backend when the network conditions are poor and the target bitrate is low (100 kbps), and 2× more frames when the network conditions are good and the target bitrate is higher (1000 kbps).

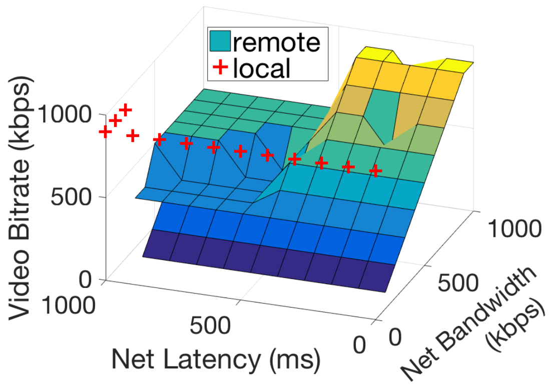
2.4.5 Performance evaluation

We now study the behavior of the DeepDecision, and compare its performance against baselines.

Different network conditions. First, we examine how our algorithm's decision changes for different network conditions (latency and bandwidth). See Fig. 2.8. As the interac-



(a) Accuracy



(b) Video bitrate

Figure 2.8: Better network conditions result in higher accuracy. To achieve good accuracy when the network latency is large, the bitrate must be carefully chosen to reduce the total transmission time.

tions between video quality (free variables), network conditions (constraints), and accuracy (objective) are complex, the decision boundaries formed by our algorithm are non-smooth and sometimes even discontinuous. In Fig. 2.8a, we plot how the accuracy of the machine learning model chosen by our algorithm changes with network latency and bandwidth. The red dots are scenarios where DeepDecision chooses to execute a model locally, and the plane represents choosing the offload. One can see that when there is no network connectivity, DeepDecision is forced to choose local models. Another scenario where DeepDecision chooses local models is when the bandwidth is non-zero but the latency is very large (1000 ms), because there is too much transmission delay to the server, resulting in stale frames and decreased remote accuracy. Note that sometimes DeepDecision prefers remote models even their accuracies are lower than local models (when latency is slightly less than 1000 ms, and bandwidth is small but non-zero). This is because backend models are faster so we can process more frames, which will increase the objective function (2.1). In general,

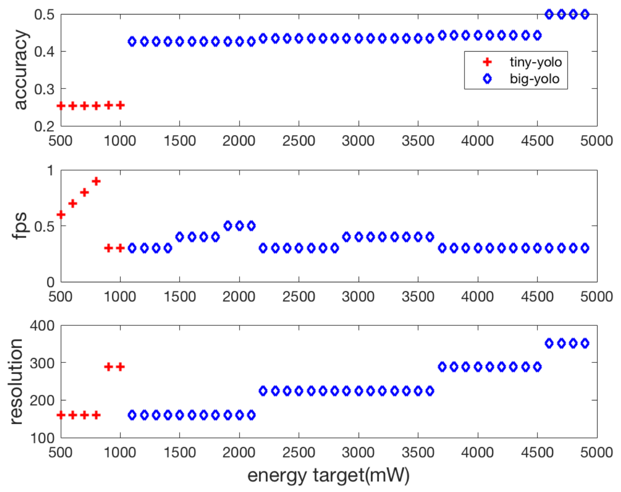


Figure 2.9: Impact of energy budget on accuracy and frame rate metrics. With additional energy budget, DeepDecision must choose which metric to increase.

when the bandwidth increases and/or the network latencies decreases, the performance of DeepDecision improves.

Fig. 2.8b shows how DeepDecision chooses the video bitrate when the network conditions change. We note that while the accuracies shown in Fig. 2.8a are deceptively smooth when we decide to offload, the bitrates chosen by DeepDecision to achieve these accuracies are highly non-smooth (as a function of network conditions) and sometimes discontinuous. In particular, the intuition is that when network latency is high ($>$ 400 ms), rather than transmitting the video at the maximum possible bitrate, DeepDecision instead (counter-intuitively) offloads at a slightly lower video bitrate in order to save network transmission time and prevent stale frames from decreasing the accuracy of the remote model (Fig. 2.7).

Energy target. The battery (*i.e.*, energy target) plays an important role when network conditions are poor (namely, very long latency or very low bandwidth). See Fig. 2.9 for how

DeepDecision makes decisions under such a harsh circumstance. This figure illustrates three major decision variables as determined by our algorithm: accuracy, frames per second (fps), and resolution. The red dots mean that the DeepDecision decides to run *tiny-yolo* locally. The blue dots mean that DeepDecision decides to execute *big-yolo* locally.

The main observation is that the energy budget needs to exceed a certain threshold in order to start using *big-yolo*. When the phone has an extremely small battery target, it is only able to execute *tiny-yolo*, which uses less energy (see Fig. 2.3b). If the battery target increases, the question is whether DeepDecision should use that extra energy to (a) increase the frame rate of the current model, (b) increase the accuracy of the current model by increasing the resolution, or (c) bump up the accuracy overall by upgrading to a more powerful model? Our results in Fig. 2.9 show that initially, DeepDecision will try to increase the frame rate while keeping accuracy unchanged. Then, with more battery, DeepDecision tries to increase resolution to allow the current *tiny-yolo* model to have higher accuracy. Finally, when the battery target is large, DeepDecision chooses the more powerful *big-yolo* model.

Comparison against baselines. In this set of experiments, we evaluate the real-time performance of DeepDecision. In our testbed, we vary the network bandwidth from 0-1000 kbps, allow the network latency to fluctuate naturally, and plot the accuracy over time in Fig. 2.10. We also plot the performance of the baseline algorithms (strawman, local-only, and remote-only).[3] DeepDecision estimates the network bandwidth and latency by

[3]Specifically, local-only runs *tiny-yolo* with resolution 160×160, and remote-only runs on the server with a video bitrate of 500 kbps. The strawman is based on [40] and uses a fixed resolution of 320×240 and a video bitrate of 500 kbps, and picks the model (local or remote) with the best accuracy.

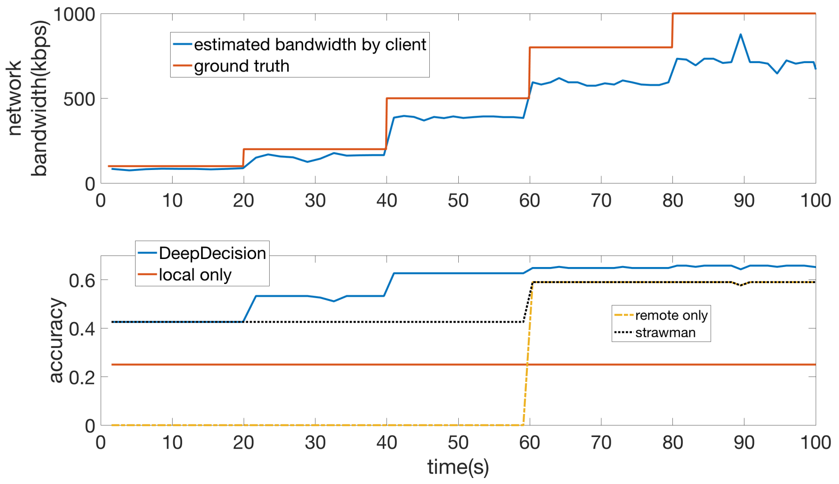


Figure 2.10: Performance of DeepDecision compared to baseline approaches. DeepDecision is able to provide higher accuracies under variable network conditions.

sending small 50 kB probe packets every second. It uses 3727mW and 60.6% CPU usage when executing remotely, and 2060mW with 38.8% when executing remotely.

Initially, DeepDecision chooses a local model, but as the network bandwidth increases over time, it switches to offloading at around $t = 20$, which boosts accuracy. Past $t = 20$, DeepDecision selects the right combination of bitrate and resolution to further maximize the accuracy. The local-only approach, on the other hand, always has a low accuracy since it uses *tiny-yolo*. The remote-only approach is not able to run initially when the network bandwidth is low. The strawman approach is slightly more intelligent; it starts offloading around $t = 60$ when the network bandwidth is high enough to support the video bitrate, but suffers from reduced accuracy before that. Moreover, since the strawman uses a fixed resolution, it does not know how to select the right combination of resolution and bitrate after it begins offloading and achieves worse accuracy than DeepDecision. Overall, the accuracy of the model chosen by DeepDecision is always higher than that of the baseline approaches. The frame rate is also high (about 15 FPS, capped by the server processing latency, which is not shown). We see that DeepDecision is able to leverage the changing of network conditions and always provide the best accuracy model to the user, by adapting the video bitrate and resolution accordingly, whereas the baseline approaches are less responsive to changing network conditions.

2.5 Related Work

Deep learning: Recently, applying CNNs to object classification has shown excellent performance [57, 83]. [82] also used CNNs to perform object detection with an emphasis on real-time performance. [46] compares the speed and accuracy tradeoffs of various CNN models. However, none of these works have considered the performance of CNNs on mobile phones. Several works have studied model compression of CNNs running on mobile phones. [47] uses the GPU to speed up latency, while [72] considers hardware-based approaches for deciding important frames. Our approach is complementary to these in that we can leverage these speedups to local processing, while also considering the option to offload to the edge/cloud.

Mobile offloading: [22] developed a general framework for deciding when to offload, while [77, 92] specifically study interactive visual applications. These frameworks cannot directly be applied to our scenario because they do not take into account that machine learning models may be compressed when executed locally as opposed to remotely. [20, 39, 94] explore remote-only video analytics on the edge/cloud, whereas we focus on client-side decisions of where to compute. Specifically, [20] considered modifying the data (sending a subset of frames) to reduce latency, while we also consider modifying the machine learning model to reduce latency. [39] offloads processing from Google Glass to nearby cloudlets. [94] performs resource profiling similar to our work, but focuses on server-side scheduling whereas we focus on client-side decisions. [79] provides some initial on-device profiling. The closest to our work is perhaps [40], which decides whether to offload CNNs to the cloud; however, they do not consider the current network conditions or profiling of video compression, energy consumption, or machine learning accuracy, which can greatly impact the offloading decision.

2.6 Summary and Future Work

In this chapter, we developed a measurement-driven framework, DeepDecision, that chooses where and which deep learning model to run based on application requirements such as accuracy, frame rate, energy, and network data usage. We found that there are various tradeoffs between bitrate, accuracy, battery usage, and data usage, depending on system variables such as model size, offloading decision, video resolution, and network conditions. Our results suggest that DeepDecision can make smart decisions under variable network conditions, in contrast to previous approaches which neglect to tune the video bitrates and resolution and do not consider the impact of latency on accuracy. Future work includes using object tracking to reduce the frequency of running deep learning, generalizing the algorithm for a larger set of edge devices, and customizing the algorithm for different categories of input videos. The hope is that architectures such as DeepDecision will enable exciting real-time AR applications in the near future.

2.7 Proof

Local $\rightarrow$ remote

Suppose we are currently using a local model, and the network bandwidth improves temporarily. Intuitively, we should only switch to offloading if the bandwidth improves a lot, or if the change will last a long time. We consider the cost incurred by switching as a time cost; specifically, upon ceasing the local model, there is a gap of time between when the last result from the local model was given, the first new result from the server arrives. This gap reduces the frame rate and the overall accuracy.

We can analyze this as follows. Let T be the length of time of the network bandwidth improvement. Let f^*_{local}, a^*_{local} be the solution of Prob. 1 assuming the model decision is fixed to be local, and $f^*_{remote}, r^*_{remote}$ be the solution rate of Prob. 1 assuming the model decision is fixed to be remote. Suppose we stay on the local model. Then the utility is $f^*_{local} + a^*_{local}$. If we switch to the remote model, then the new frame rate is $f^*_{rem} - \frac{(\frac{rf_0}{B}+L)f^*_{remote}}{T}$, where the second term represents the frames not processed during the switching process. Rearranging terms, we can find the condition to switch from the local -> remote model:

$$a^*_{local} - a^*_{remote} \leq f^*_{remote}\left(1 - \frac{1}{T}\left(\frac{r^*_{remote}f^*_{remote}}{B} + L\right)\right) - f^*_{local} \tag{2.14}$$

Intuitively, this shows that the factors that contribute to switching are: long length of time T of improved bandwidth, bandwidth B is large, and network latency L is low. The algorithm should only switch if condition (2.14) is satisfied.

Remote $\rightarrow$ local

We now analyze the opposite case, where the algorithm may currently be using a remote model, the network conditions decrease, and the algorithm considers switching to a local model. However, if the time T of the poor network conditions is small, then it may not be worth switching. This case is different from the local-¿remote case because the switch is basically instantaneous: the algorithm stops sending to the remote server, and can immediately start using the local model, without any connection setup time. Analytically, the condition for switching from remote to local is:

$$f^*_{rem} - f^*_{loc} \leq a^*_{loc} - a^*_{rem} \tag{2.15}$$

Considering both switching conditions (2.14) and (2.15) together, we realize something interesting. There is some kind of inherent stability. In general, it is likely that the frame rate of the server model will be greater than the frame rate of the local model: $f^*_{rem} > f^*_{loc}$. It is also likely that the accuracy of the server model will be greater than the accuracy of the local model: $a^*_{rem} > a^*_{loc}$ Condition (2.14) to switch from local to remote would be easily satisfied, except for the "network conditions term", $\left(1 - \frac{1}{T}\left(\frac{r^*_{rem} f^*_{rem}}{B} + L\right)\right)$. Condition (2.15) to switch from remote to local would be difficult to satisfy, but at least it does not have the disadvantage of the network conditions term. So given the current decision, the system does not easily switch to another decision, meaning the system has some kind of inherent stability.

Chapter 3

Change Detector for Power Thrifty Object Detection and Tracking

3.1 Introduction

AR is popular in the market today [66] with potential applications in many fields including training, education, tourism, navigation, and entertainment, among others [18]. In AR, the user's perception of the world is "augmented" by overlaying virtual objects onto a real-world view. These virtual objects provide relevant information to the user and remain fixed with respect to the real world, creating the illusion of seamless integration. Examples of AR apps used today include Pokemon Go, Google Translate, and Snapchat filters.

An important task in the AR processing pipeline is the detection and tracking of the positions of real objects so that virtual annotations can be overlaid accurately on top [54, 20, 61]. For example, in order to guide a firefighter wearing an AR headset, the

AR device needs to analyze the camera frame, detect regions of interest in the scene (e.g., victims to be rescued), and place overlays at the right locations on the user display [74]. Commercial AR platforms such as ARCore and ARKit can understand the 3D geometry of the scene and detect surfaces or specific instances of objects (e.g., a specific person), but lack the ability to detect and track complex, non-stationary objects [36, 61].

To track real objects, AR apps can use tracking by detection techniques [86], wherein each camera frame is examined anew to detect and recognize objects of interest; both object locations (*e.g.*, bounding boxes) and class labels are output. Tracking by detection is used, for example, by the open-source ARToolKit [1] to track fiducial markers in the scene. To go beyond this to detect non-fiducial objects in the scene being viewed, one can employ state-of-the-art DNN-based object detectors which yield high object recognition and detection precision (with regards to objects in general). However, a naive plug and play of DNN-based object detection and recognition into a tracking by detection framework will exacerbate the already high battery drain of mobile devices, which is of great concern to mobile users [42]. While the screen, camera, and OS do consume a large portion of the user's battery (3-4 W in our measurements), continuous repeated executions of DNNs (even those models optimized for mobile devices, e.g., [82, 45]) will also consume a major portion (1.7-3 W) of the battery.

Recent works have targeted improving the energy efficiency of DNNs (*e.g.*, by using specialized hardware [47] or via model compression [41]); however, they focus on individual DNN executions on individual input images [47], rather than understanding energy consumption across time, as is needed in AR or other continuous tracking applications.

Invoking DNN executions on every captured frame in an AR application will cause high energy expenditure even with such mobile-optimized methods.

In this chapter, we ask the question: How can AR apps achieve good object detection and tracking performance and yet consume low energy? To answer this, we make the key observation that while using a DNN is important for detecting new objects, or when significant changes to a scene occur, lightweight incremental tracking can be used to track objects otherwise, in between DNN executions. This saves precious computation and energy resources, but requires initial knowledge of the object to be tracked (which must be supplied by the DNN). To realize such an approach, however, a key question that needs to be answered is "when should DNNs be invoked and when is incremental tracking sufficient to maintain similar accuracies as the DNN?" Although tracking by detection and incremental tracking have been studied together to a limited extent [100, 56], these prior approaches either trigger the DNN at a very high frequency (*e.g.*, every 10 frames), use heavyweight object trackers, and/or assume complete offline knowledge of the video. These limitations make such methods inappropriate for real-time AR applications and/or mobile platforms with battery limitations.

As our main contribution, we design and implement MARLIN (Mobile Augmented Reality using LIghtweight Neural network executions), a framework that addresses the critical problem of limiting energy consumption due to object tracking for AR, while preserving high tracking accuracy. Specifically, MARLIN chooses between DNN-based tracking by detection and incremental tracking techniques to meet three goals: (a) good tracking performance, (b) very low energy drain, and (c) real-time operations. Briefly, MARLIN first

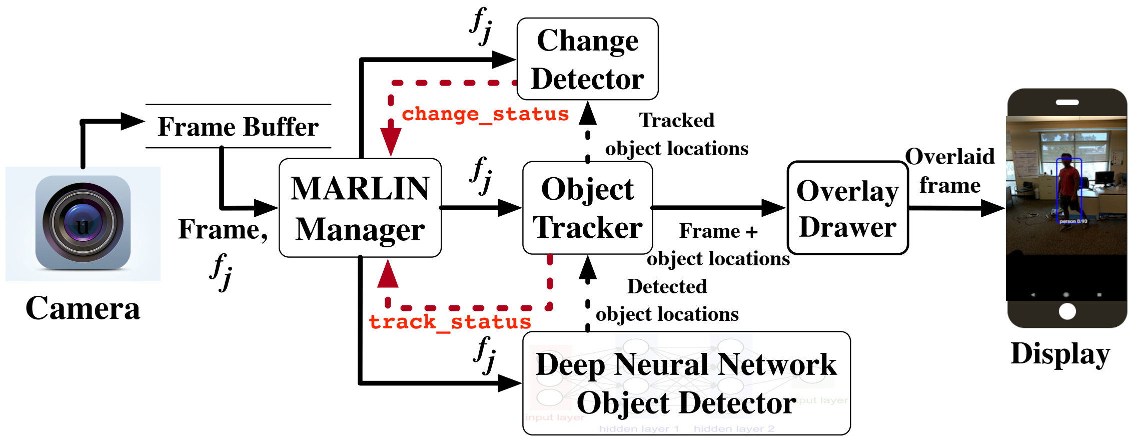


Figure 3.1: Overview of MARLIN's architecture

performs DNN-based tracking by detection on an initial incoming frame to determine the object locations. Once such objects are detected, MARLIN performs incremental tracking on them to continuously update the locations of the relevant AR overlays; the tracker also checks every frame for significant changes to the object (e.g., a car door opening) to determine if tracking by detection needs to be re-applied. In addition, MARLIN employs a novel change detector that looks for changes to the background (e.g., appearance of new objects) that are likely in the AR scenarios of interest.

In this chapter, we will talk about the overview of MARLIN and the design and performance of its core function, change detector.

3.2 MARLIN System Overview

Fig. 3.1 provides an overview of MARLIN's architecture, composed of pipelined operations from a camera (left) to a display (right). The input to this pipeline is a frame from the camera and the output is a view with overlaid augmented objects (specifically, overlaid bounding boxes in this work) on top of the physical objects (e.g., a person). Each input frame from the camera is buffered before being fetched by the "MARLIN Manager" module. MARLIN Manager is a real-time scheduler that assigns each incoming frame to one or more of the three modules viz., the object tracker, the change detector, and the DNN object detector. These modules act as workers for MARLIN Manager, i.e., each module only processes frames that are assigned to it by MARLIN Manager.

By default, MARLIN Manager assigns a new frame to the object tracker. The object tracker updates the locations of the objects from the previous frame to the current one. In addition to tracking objects, it returns a "track status" which indicates the fidelity of tracking and alerts MARLIN Manager of any changes to the current set of tracked objects.

To check for new objects in a scene (that require tracking), MARLIN Manager assigns an input frame to the change detector module. In addition to this input frame, the change detector needs to know the locations of the current set of tracked objects (so that those are ignored). By ignoring objects that are already tracked with high accuracy by the object tracker, the change detector avoids unnecessary alerts. It only analyzes the parts of the frame that are "external" to the current set of tracked objects, and issues an alert to MARLIN Manager if there are significant changes in these parts.

MARLIN Manager only sends a frame to the DNN object detector if it needs to detect and classify new objects in that frame, or when features relating to currently tracked objects change significantly. This is because the DNN is the most energy-draining module in MARLIN and must only be invoked on a need to basis. It (MARLIN Manager) uses tracking information and the output of the change detector to determine if the frame should be assigned to the DNN. Finally, the object tracker provides information that specifies the object locations and the class labels to the "overlay drawer." The latter draws virtual overlays (bounding boxes) on top of the actual objects in the frame and forwards the augmented frame to the display.

3.3 Lightweight Change Detector

3.3.1 Overview of Change detector

While the object tracker tracks stable objects and triggers a DNN only when significant changes occur relating to these (i.e., a person's posture changes by quite a bit), MARLIN must also be able to handle new objects that appear in the scene (e.g., a person appears). To this end, we design a change detector which detects changes *not* pertaining to the objects already being tracked (i.e., new objects coming into view). The key challenge in designing such a change detector is avoiding high false positives with respect to previously tracked objects (causing extraneous DNN executions). However, our experiments with existing approaches [9, 102, 101] show high false positive rates of approximately 20-100%, resulting in numerous unnecessary DNN executions consuming high energy, even on a simple video with one slowly moving object and a moving camera (detailed results omitted due to space). Towards preventing such false positives, our key idea is to "hide" existing objects from the change detector by changing the corresponding pixels to a common value, whose value does not change across frames.

Functional description: When the change detector receives a frame (and the locations of currently tracked objects) from MARLIN, it converts the frame into a feature vector via the following steps: **(i)** It first colors all rectangular boxes corresponding to the locations of the currently tracked objects white (maximum pixel intensities for red, green and blue channels) to generate what is called a COLORED_IMAGE; **(ii)** It resizes this to 128×128 pixels to form a new image (RESIZED_COLORED_IMAGE), and also calculates the histograms of the red, green, and blue channels of RESIZED_COLORED_IMAGE; **(iii)** Finally, it recasts RESIZED_COLORED_IMAGE, which is a 2D array of pixels, into a single row vector, and appends the three histograms to the end of the row (resulting in another row vector). Thus, it converts an input image of size 640x480x3 (width, height, channels) into a feature vector of size 1x49920 of floating point numbers. This means that we compress it by a factor of 18 (from 921,600 to 49,920 numbers) because we want to quickly perform change detection and do not need all information contained in the frame. Specifically, we focus on the color features and do not use other features such as keypoints, which we experimentally found to be computationally expensive (also shown in [23]).

We reiterate that any changes to tracked objects (now "whited out" in step (i) above) are handled by the object tracker. To detect changes external to these objects, the change detector uses a random forest classifier with the color features as the input vector. The forest consists of 50 decision trees (total 55,796 nodes). Each (binary) tree has a maximum depth of 20 and each node in the tree is a logical split that takes a variable (an element in the feature vector) and checks its value against a threshold that was learned during model training. These thresholds represent natural colors of backgrounds (e.g., sky or grass or whited-out pixel) and foregrounds (e.g., tiger or elephant) in order for each node to decide whether or not this frame contains a significant change. The output of each tree is obtained by reaching a leaf node (after moving through splits down the tree) and the final detection result is by a majority vote across all the trees. We also tried other lightweight classifiers such as Support Vector Machines, but found experimentally that random forest had the highest change detection accuracy.

Runtime execution: MARLIN invokes the change detector after the object tracker, which provides the updated objects' locations in the current frame. The change detector then uses the supervised classifier to detect changes to the input feature vector. It inputs the above feature vector to the classifier and outputs 1 (change detected) or 0 (no change detected).

Exceptions: In most cases, the change detector reports a change prior to the handling of the subsequent frame. If in the rare case, the change detector finishes its checks after a subsequent frame arrives, the change detection result will be used by MARLIN to trigger the DNN (if needed) as soon as the result is received.

3.3.2 Change detector model training

The change detector is implemented as a random forest classifier trained with 100,000 video frames from the ImageNet dataset. Because the video clips were of different lengths, to avoid biasing the change detector towards longer videos, we randomly chose 30 frames from each video for training. The training set is divided into four subsets: (1) unmodified frames with at least one new object (`change_status` is true); (2) frames with existing tracked objects colored white but with at least one new object in the background (`change_status` is true); (3) frames where all objects in the scene were already tracked and colored white (`change_status` is false); (4) unmodified background frames with nothing else (`change_status` is false). This labeling resulted in 50% of the training set being labelled with `change_status` is true and the other 50% labeled as `change_status` is false.

We experimented with various classifiers (random forest, support vector machines, shallow neural network), and with other input features (e.g. edges, colors, histogram of gradients). On the 10,000-frame validation set, the random forest classifier using color histogram and pixel input features achieved the best performance across all tested models, with 88.0% precision and 81.7% recall on the binary classification task. In comparison, e.g., SVM using HOG features has 64.9% precision and 61.4% recall.

3.4 Related Work

Using the sum of absolute differences is a naive method of change detection, and is susceptible to noise from illumination or background changes [9, 78]. Background/foreground subtraction methods using GMM [101] and KNN [102] are more robust, but assume static cameras, which is not true for AR. Alternatively one could use object detection to check if there are changes over time (e.g. [30]); however, the feature extraction step of such methods are heavy-weight and unsuitable for mobile devices.

3.5 Summary

In this chapter, we developed a novel lightweight change detector that looks for video background changes. It has very high precision(88.0%) and recall(81.7%), and is applied to trigger DNN detection. When DNN is not triggered, lightweight tracking algorithms will be applied to save energy. Our experiment shows the change detector has a very low power consumption of 0.1W and a very low latency of 4ms per frame.

Chapter 4

Multi-User Augmented Reality with Communication Efficient and Spatially Consistent Virtual Objects

4.1 Introduction

Augmented Reality (AR) applications have recently exploded in popularity among smartphone users. In AR, a user's field-of-view (FoV) is overlaid with virtual objects, which should remain fixed with respect to the real world in order to provide a seamless transition between the real world and the virtual objects. As AR becomes more popular, a natural question is: can we share the virtual objects with other users? Based on the off-the-shelf

AR apps currently available, the answer is “yes”. For example, Pokemon Go released the Buddy Adventures feature in December 2019, which allows multiple users to view their virtual creatures together in the same real world space. Other multi-user AR applications currently available include Just a Line, where multiple users can collaboratively draw virtual graffiti, and Minecraft, where users can build structures together from virtual blocks.

Yet despite their emerging popularity, little is known about the network communications of multi-user AR apps. The fact that these apps involve multiple users clearly indicates that some form of network communication is required. However, it is currently unknown how these apps communicate, what they are communicating, and how the data communications impact user experience. This work seeks to address this key gap in knowledge, and propose solutions to the problems that we find in this space.

Through our measurements of off-the-shelf multi-user AR apps (detailed in Sec. 4.2), we find that users experience multiple seconds of latency between one user placing a virtual object to it appearing on another user’s display. Moreover, the virtual objects can appear at different locations, with respect to the real world, on each user’s display. These two problems – latency and spatial inconsistency – are key factors in AR user experience [98, 58, 21], and we find that they depend on the communicated information between the AR devices. Thus the goal of this work is to optimize the network communications of multi-user AR apps, to enable fast and accurate coordination of the virtual objects across AR displays. For example, in a classroom, students equipped with AR devices should be able to see the same virtual chemistry molecule and manipulate it, with the virtual molecule remaining consistent across students. However, uncoordinated or laggy updates to the virtual molecule would break the illusion of seamless integration with the real world, and result in artifacts such as other users appearing to touch non-existent parts of the virtual molecule.

We meet and address several technical challenges towards realizing this vision. (1) Firstly, multi-user AR apps share a large initial data burst containing information about the real world environment, in order to render the virtual objects at the correct locations on each AR display. This leads to multiple seconds of latency whenever users move to a new area and wait for virtual objects to appear. *To address this, we propose communication strategies that adapt to the positions of the virtual objects in order to reduce communication latency.* (2) Next, as the user moves around, the AR device continuously observes new information about the real world. These observations can be processed to update the virtual objects' positions with respect to the real world, but the update may actually harm the positioning accuracy if the wrong information is used. *To address this, we propose a new "feature geo distance" metric that allows the AR app to select the right camera frames to re-align the virtual objects' positions with other users.* (3) Finally, multi-user AR apps do not know if their virtual objects are drifting in time or in space. Without knowing that the virtual objects are experiencing positioning issues, these apps does not know when problems need to be corrected. *To address this, we develop a methodology and tool to automatically quantify spatial inconsistency issues of the virtual object across time and across users.*

While there has been research on object detection and cloud/edge offloading for AR (*e.g.*, [53, 61, 80]), these works typically consider a single user viewing the virtual objects, rather than multi-user coordination. They also do not incorporate simultaneous localization and mapping (SLAM), which is part of off-the-shelf AR systems today to en-

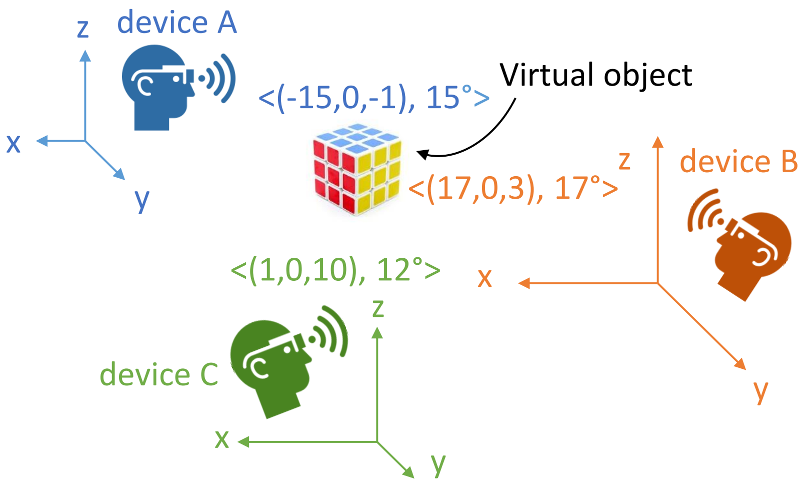


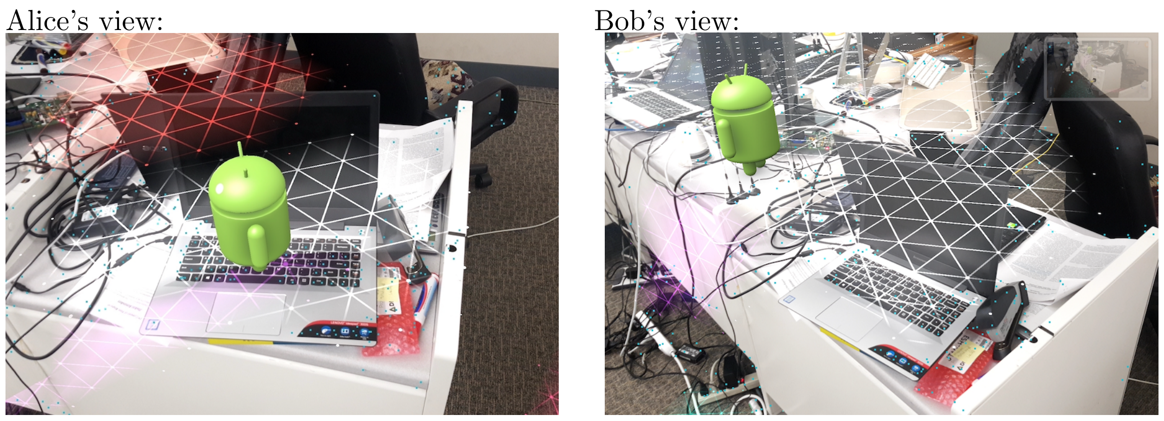
Figure 4.1: Multiple AR devices try to ensure consistent views of a virtual object, despite different coordinate systems.

able 3D understanding, but contribute to the latency and spatial inconsistency issues that we observe. Industry players have started to look at multi-user AR [35, 67, 14], but focus mainly on application development and not the communication aspects of the underlying platform. To the best of our knowledge, no prior work has systematically examined AR spatial consistency issues when there are multiple users, and their dependency on the network data transmissions between the devices.

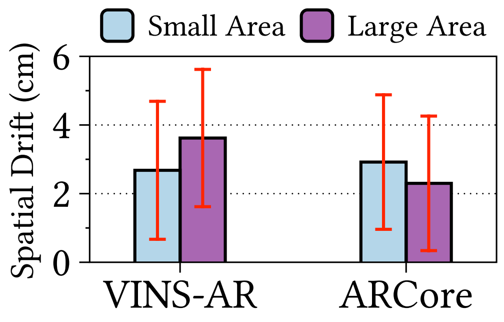
In summary, we study the emerging application of multi-user AR and its networking aspects. In particular, we focus on rendering virtual objects that are clustered around a common "anchor" point in the real world [35]; this is common in typical multi-user AR apps (*e.g.*, all the virtual Pokemon or virtual graffiti are placed near each other). We call our system SPAR (SPatially Consistent AR). Our contributions include:

- We measure the performance and network usage of off-the-shelf AR apps, and identify problems of high latency, spatial drift, and spatial inconsistency of the virtual objects over time and across users.
- We develop new methods for efficient communication and computation of the AR devices. Specifically, SPAR performs the following: (i) it adapts the communicated information to the virtual object positions, to reduce latency; (ii) it efficiently computes the virtual objects' positions in each user's display through lightweight coordinate system alignment, to create spatial consistency; (iii) it continuously updates the positions of the virtual objects with respect to the real world using a new update metric, to maintain spatial consistency; and (iv) it automatically quantifies the spatial drift and consistency of the virtual objects, to evaluate performance.
- We perform evaluation on Android AR devices, extending VINS [60], a 6DoF-based AR platform, with multi-user capabilities. We work with open-source systems because existing AR platforms from Android, Apple, and Microsoft [35, 14, 67] are closed source and thus their internal code cannot be modified for experimentation. Our results show that SPAR can decrease the total latency by up to 55%, while decreasing the spatial inconsistency of the virtual objects by up to 60%, compared to a baseline method of communicating the full data or off-the-shelf AR platforms. We also show that our tool can estimate a virtual object's spatial drift and inconsistency with a low RMSE of only 0.92 cm, compared to manual human labeling.

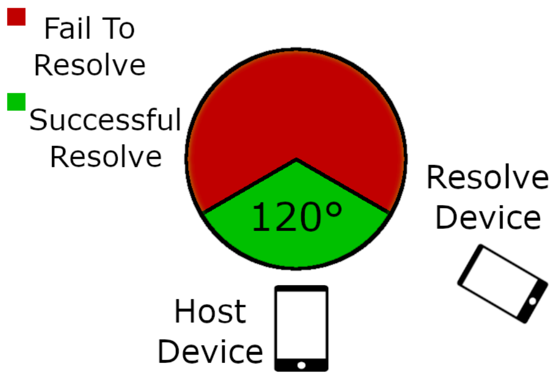
In the remainder of this chapter, we discuss the measurements that motivate this work (Sec. 4.2), a brief background (Sec. 4.3), the overall system architecture (Sec. 4.4), and the



(a) Spatial inconsistency across two AR devices.



(b) Spatial drift



(c) Spatial inconsistency

Figure 4.2: Quantifying spatial drift and inconsistency. The resolving devices suffer from spatial drift of 2-4 cm and fail to resolve if their orientation with respect to the host is more than 60°.

design of the individual components (Sec. 4.5). We then evaluate the system (Sec. 4.7), discuss related work (Sec. 4.9), then conclude (Sec. 4.10).

4.2 Motivation: Spatial Drift, Inconsistency, and Latency of Off-the-Shelf Mobile AR

To showcase the issues of virtual object spatial inconsistency and latency in multi-user AR, we examined several off-the-shelf AR Android apps. The four Android apps we tried were Google CloudAnchor [35], Google Just a Line [37], Minecraft [69], and Pokemon Go [73], running on Pixel 4 smartphones. We have two users, Alice and Bob, who perform the following sequence of user interactions:

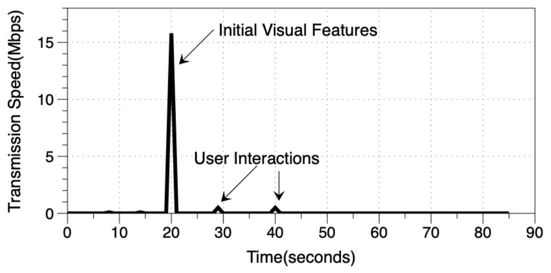
1. *Place initial virtual object:* Alice places a virtual object by tapping the screen.
2. *Render initial virtual object:* Bob waits for it to appear on his screen.
3. *Subsequent user interactions:* Alice taps the screen to move a virtual object or place new virtual objects (only Just a Line and Minecraft have this feature). Bob waits for the update to be reflected on his screen.

Our main finding is that there is significant delay and spatial inconsistency between when Alice places the first virtual object and Bob renders it on the screen (steps 1 and 2 above), thus motivating the work in this chapter. Similar results hold when there are multiple receiving users (*i.e.,* multiple Bobs) who wish to view Alice's virtual objects. Below, we detail our quantitative measurements.

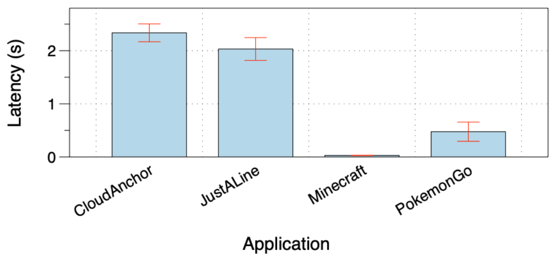
Spatial drift and inconsistency: We experimented 5 times with CloudAnchor, with each trial lasting 1 minute with Bob moving 1 m (small area) to 4 m (large area) in the real world. In these experiments, we observed two types of spatial issues relating to the virtual object's position:

- *Spatial drift:* For a single user, the virtual object can drift in position over time.
- *Spatial inconsistency:* When there are multiple users, the virtual object can appear in a different location, relative to the real world, on each user's display, as shown in Fig. 4.2a.

In the single-user case (Fig. 4.2b), the spatial drift is around 2-3 cm in both the large and small areas, for both ARCore [32] and VINS-AR [60] platforms. In the multi-user case, spatial inconsistency results are larger (results later in Sec. 4.7), and moreover we observe that when the two devices are placed more than 60° apart, the virtual object fails to resolve, as shown in Fig. 4.2c. We also observed similar issues with Magic Leap, but focus on smartphone-based AR in this work due to the ubiquity of smartphone devices. These spatial drift, inconsistency, or failure to resolve a virtual object can cause serious issues when multiple users interact in a shared AR session (*e.g.*, multiple users jointly building a tower of building blocks), and are key contributors to user experience in AR [98, 58]. This motivates SPAR's goal of reducing spatial drift and inconsistency. Similar to prior work [89], we focus on positioning errors, as we did not observe much rotational error our experiments. The low rotational error we observe may be due to our use of visual-inertial SLAM as the basis for our AR system, which generally has less rotational error than pure visual SLAM [55].



(a) Just a Line network trace.



(b) Communication latency.

Figure 4.3: AR apps transmit a large amount of environment data for the initial virtual object, resulting in mult-second communication latency.

Additionally, from running these experiments, we found that using absolute trajectory error to evaluate SLAM accuracy doesn't fit for AR applications, and manually measuring the position of the virtual object is laborious and time-consuming, with each measurement requiring several seconds per frame for a human to examine the image and record the virtual object's position. This motivates SPAR's spatial drift and inconsistency estimation tool, to reduce the amount of human labor and enable AR apps to automatically quantify these issues.

Communication latency: We next examine the data transfer between the AR users. Fig. 4.3 shows a representative network trace from Just a Line (other apps have a similar pattern, not shown for brevity). We observe a large data transfers for the initial object placement (steps 1 and 2), 9-20 Mb, and smaller data transfers during the subsequent user interactions (step 3), < 100 kb. These large initial data transfers prolong the communication latency. With an upload bandwidth of 8 Mbps (the average in our lab), the initial communication latency is plotted in Fig. 4.3b. The communication latency of > 2 s for the ARCore-based apps (CloudAnchor and Just a Line) is a lower bound on the user-experienced delay (while Minecraft and Pokemon Go have lower communication latency, they suffer from higher spatial inconsistency). This latency is a key contributor to user experience in multi-user AR [98, 21], thus motivating SPAR's goal of reducing user-perceived latency.

Connection between latency and spatial inconsistency: We discover there is a critical connection between the aforementioned spatial inconsistency and latency issues we observed. Off-the-shelf AR platforms use similar methods [35, 14, 67] to share real world environment information between users during step 1, resulting in the large spikes of data observed in Fig. 4.3. This environmental information helps align the coordinate systems of the AR devices, enabling each device to render the virtual objects at fixed positions in the real world. In other words, the data communicated during the placement of the initial virtual object (step 1) directly impacts the spatial drift and inconsistency. This motivates SPAR's communication-efficient strategies to decrease communication latency, while trading off with spatial drift and inconsistency.

4.3 Brief Background on AR

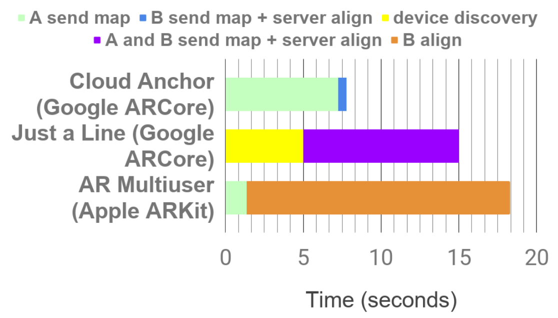
Current AR platforms such as Google ARCore, Apple ARKit, and Microsoft Hololens rely on simultaneous localization and mapping (SLAM). SLAM solves the problem of when a device is in an unknown environment, how to build a consistent map and localize itself at the same time [25]. In a typical single-user AR scenario, the AR device uses SLAM to construct a *point cloud* representing the 3D coordinates of features in the real world, and also estimates its own location and orientation (known as *pose*). To compute the point cloud, SLAM selects a subset of camera frames (known as *keyframes*), extracts features from the keyframes, runs SLAM algorithms on the features, and outputs the 3D coordinates of the features in each keyframe (*i.e.,* the point cloud) and the estimated device pose. These SLAM algorithms includes keyframe matching to localize the device with respect to its past trajectory. Each keyframe data structure contains feature descriptors, 2D feature coordinates with respect to the camera image, and 3D feature coordinates with respect to the real world. The 3D feature coordinates are relative to an origin point in the real world, which is called the device's *world coordinate system.*

To render virtual objects for AR, the device records the pose of the virtual object (defined as its location and orientation, which can be provided by user input or by an object detector). The AR device runs SLAM continuously to update its own pose estimate and the 3D coordinates of the features in the point cloud, and then draws the virtual object on the display when its FoV overlaps with the virtual object's pose.

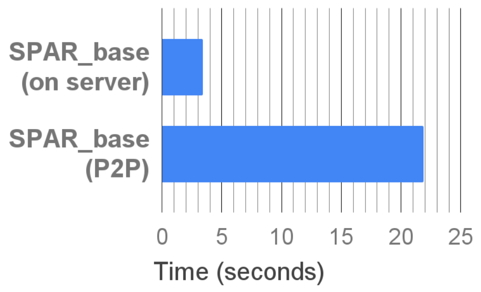
4.4 System Architecture

4.4.1 System Architecture of Existing Applications

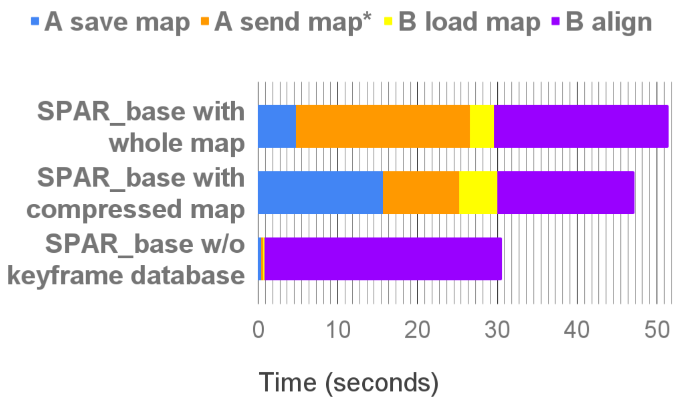
There are two primary communication architectures in current SLAM-based mobile AR systems: cloud-based and P2P-based. We describe a hosting device (Alice, or A, the "host") who places the virtual objects, and a resolving device (Bob, or B, the "resolver") who wishes to view the same virtual objects. Alice places a virtual object in its environment, and wishes to share this information with a newly joined device, Bob.



(a) End-to-end latency of current mobile AR apps.



(b) Time to align coordinate systems on a server versus mobile device in SPAR_base.



(c) SPAR_base (P2P version) reduces latency by optimizing data transmissions.[1]

Figure 4.4: Latency breakdown starting from device A placing a virtual object to device B rendering it on the display.

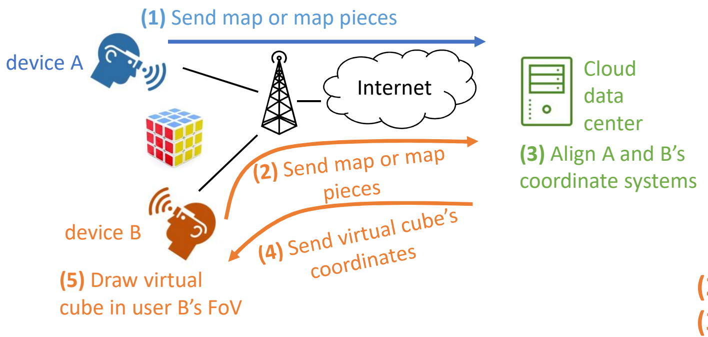

(1) Send map or map pieces
device A
Internet
Cloud data center
(2) Send map or map pieces
(3) Align A and B's coordinate systems
device B
(4) Send virtual cube's coordinates
(5) Draw virtual cube in user B's FoV

Figure 4.5: Cloud-based architecture, similar to Google ARCore.

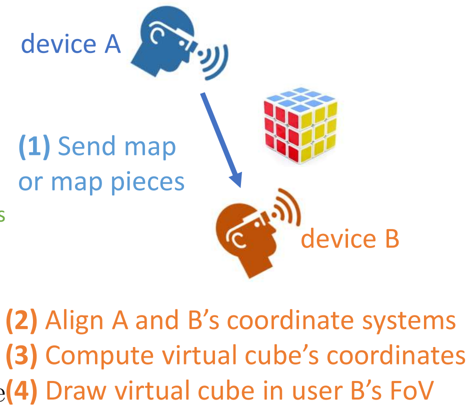

device A
(1) Send map or map pieces
device B
(2) Align A and B's coordinate systems
(3) Compute virtual cube's coordinates
(4) Draw virtual cube in user B's FoV

Figure 4.6: P2P architecture, similar to Apple ARKit.

Cloud-based: In a centralized architecture, the cloud collects device pose information from the AR devices, performs processing, and returns results as needed. Cloud-based architectures are used, for example, by Google ARCore [35] and MARVEL [19]. The information exchange is illustrated in Fig. 4.5, and described below:

1. *Device A sends*: A sends its SLAM map (or the related camera frames), and the virtual object's coordinates to the cloud.

2. *Device B sends*: B sends a piece of its map corresponding to its current location (or the related camera frames) to the cloud.

3. *Cloud aligns coordinate systems*: The cloud runs SLAM (if camera frames only were sent), then aligns A's map and B's map piece, and computes the virtual object's pose in B's coordinate system.

4. *Cloud sends virtual object's coordinates*: The cloud sends the computation result to device B.

5. *B draws virtual object*: B draws the virtual object in its world coordinate system.

P2P-based: In a de-centralized or P2P architecture, AR devices communicate directly with each other, without the assistance of a central entity. Such an architecture is followed, for example by Apple ARKit [14]. The process is illustrated in Fig. 4.6 and described below:

1. *Device A sends*: A sends its SLAM map (or related camera frames)and the virtual object's coordinates to B.

2. *Device B aligns coordinate systems*: B runs SLAM (if only camera frames were sent), then aligns A and B's coordinate systems, and computes the virtual object's pose in B's coordinate system.

3. *B draws virtual object*: B draws the virtual object in its world coordinate system.

In summary, these two architectures require similar types of computation, but at different locations (*i.e.,* on the device or in the cloud), which impacts the information exchange between the devices.

Measurements: We conducted measurements of several AR applications utilizing the above architectures in a controlled lab setting. We used Samsung Galaxy S7 smartphones with WiFi connectivity (50 Mbps download and upload speed), unless otherwise mentioned. Each measurement was repeated 3 times, with the averages plotted. In Fig. 4.4a, we show end-to-end latency measurements of three AR apps: CloudAnchor [33] and Just a Line [37] (Google ARCore demo apps), and AR MultiUser [15] (Apple ARKit demo app). We observe that latencies between device A placing a virtual object and device

B drawing the virtual object are quite long, from 7-18 s. The cloud-based apps tend to have longer communication times and shorter coordinate system alignment times, because of cloud compute resources. On the other hand, the P2P app has shorter communication time but longer coordinate system alignment time due to running the joint computations on the mobile device (in this case an iPad), leading to longer end-to-end latency overall.

Comparing Architectures in SPAR baseline

Given our understanding of the above architectures, which architecture is more suitable in different scenarios? The P2P architecture has advantages in terms of scalability (there is no central bottleneck link), and privacy (information doesn't need to be sent to the cloud). However, updates to virtual objects can take time to propagate across the devices, potentially resulting in inconsistent information. The cloud architecture has advantages in terms of compute power, and can synchronize updates about the virtual objects across devices, but relies on Internet connectivity. Another possibility is a hybrid architecture, where the devices share information only amongst themselves, but one device acts as a "master" node that undertakes communication and computation efforts. Such an approach essentially assigns one of the devices to take the role of the cloud, but places heavy computation and communication demands on the master node.

Given that these architectures are currently implemented in different mobile OSes (iOS and Android) and are closed source, an apples-to-apples comparison of their system performance metrics cannot be made between them, nor can modifications be made. Our idea is to develop an open-source reference system that allows comparison of the different communication architectures. To do this, we build on an open-source state-of-the-art

SLAM system for single users [76], and add multi-user capabilities and the ability to switch between the communication architectures observed in the commercial mobile AR platforms. This will enable accurate measurements of the performance of each component of the AR computation and communication pipeline. Our system, SPAR, will provide researchers and developers with insight into the system requirements of multi-user mobile AR, guidelines on architectural decisions, and understanding of which parts of the AR pipeline can be optimized.

We first implemented an initial prototype of SPAR with two Android devices and we call it SPAR_base, which is P2P based. At this time, no optimization is done to SPAR_base, i.e., device A sends all the generated environment information(map) to device B and device B scans the data to find a match and compute the transformation between A and B. SPAR_base allows device A to place a virtual cube and device B to receive and render the cube in its FoV, with full control over all components of the system, including SLAM algorithms, communication protocol, communication frequency, coordinate system alignment frequency, etc. In Fig. 4.4b, we show some initial measurements of SPAR_base, comparing the computation time of coordinate system alignment of P2P and server-based architectures. We can see that the edge server-based computation time is lower, suggesting that an edge-server based architecture may be ideal as communication latency is also low in edge scenarios [85]. Specifically, the map alignment time with the P2P architecture is seven times longer than in the server-based architecture, which suggests great potential for both the edge- and cloud-based architectures. In fact, the server-based architecture can still tolerate an additional 10 seconds of communication latency and still have lower end-to-end latency than the P2P architecture (Fig. 4.4b).

Black box testing: To ensure that the results produced SPAR_base reflect existing AR systems, we will perform black box testing. We will choose a set of sample AR apps and scenarios, and tune SPAR_base until its results are similar to those observed in commercial platforms (*e.g.*, Google ARCore). While we cannot have perfect reproduction of commercial platforms, due to their opaqueness, we explicitly try to match their performance for a given set of test cases. In our initial results with SPAR_base, the computation latency of the server-based architecture (upper bar in Fig. 4.4b) is roughly comparable to computation latency of Google ARCore's server-based architecture (the "B send map + server align" bar in Fig. 4.4a). (SPAR_base is slightly slower because its computation latency includes the DBoW generation time, which we were not able to measure in ARCore because it is closed-source.) Similarly, SPAR_base's P2P computation latency (lower bar in Fig. 4.4b) is comparable to ARKit's P2P computation latency ("B align" in Fig. 4.4a).

4.4.2 System Architecture Design

We designed SPAR to optimize the performance of SPAR_base. SPAR consists of the following components that work together in concert, as shown in Fig. 4.7. As a reminder, for illustrative purposes, we describe a hosting device (Alice, the "host") who places the virtual objects, and a resolving device (Bob, the "resolver") who wishes to view the same virtual objects. When there are multiple resolvers (*i.e.,* multiple Bobs), each Bob performs steps 2 and 3 below.

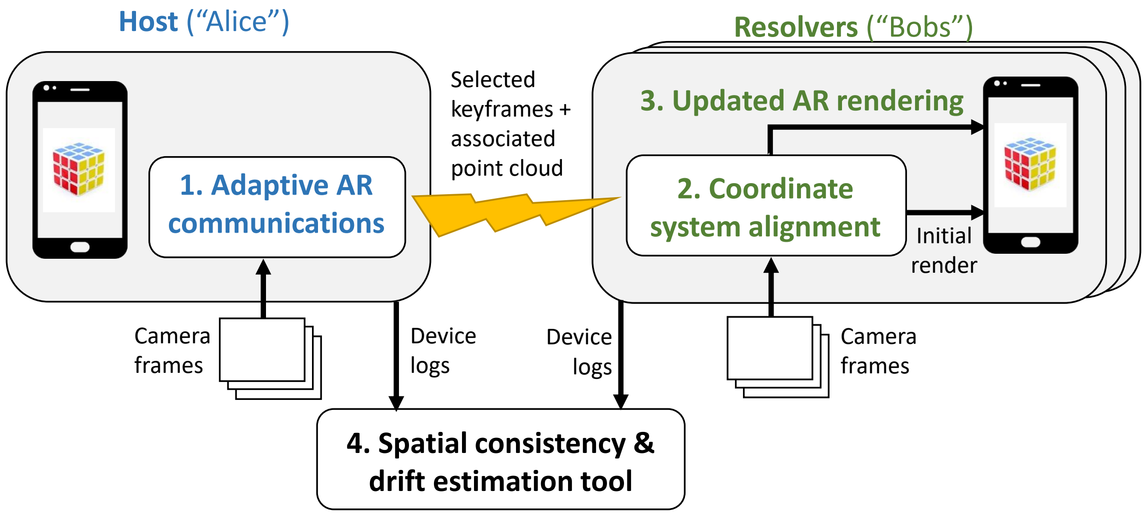

Host ("Alice")
Resolvers ("Bobs")
1. Adaptive AR communications
Selected keyframes + associated point cloud
3. Updated AR rendering
2. Coordinate system alignment
Initial render
Camera frames
Device logs
Device logs
Camera frames
4. Spatial consistency & drift estimation tool

Figure 4.7: SPAR system overview. The modules work together for fast multi-user AR with spatially consistent virtual objects.

(1) Adaptive AR communications: A naive approach to share AR information between Alice and Bob would be sending all of Alice's SLAM data to Bob(s). However, this approach is not cognizant of AR, and can be very slow; To reduce this communication latency, we design an intelligent mechanism that adapts the data transmissions to the content of the AR scene. Specifically, only areas of the real world in which Alice was near the virtual objects, and the virtual objects were visible, are sent to Bob(s).

(2) Coordinate system alignment: Using the received information, Bob next desires to render the virtual objects at the correct locations in the real world. However, since Bob and Alice can open the AR app from different positions in the real world, they lack a common frame of reference to accurately describe the pose (location and orientation) of the virtual objects. Current single-user AR does not have this problem because there is only one device and thus one world coordinate system. We propose a lightweight coordinate system alignment method, which allows Bob to align himself inside Alice's coordinate

system. The key idea is to reuse some functionality from single-user AR for path loop detection [76], which that is already running on the device, making our method very lightweight yet accurate.

(3) Updated AR rendering: After the initial rendering of the virtual objects, Bob may continue to move around the world. As Bob observes new information about his environment, he can use this information to re-align his coordinate system with Alice and update the positions of virtual objects. But how can Bob know if these updated positions are more or less accurate? To enable Bob to predict whether the spatial inconsistency improved with the new observations, we propose a new metric, *feature geo distance*, which has good correlation with spatial inconsistency. The intuition behind this metric is that if Bob observes areas close to the virtual objects, these observations can likely improve spatial consistency.

(4) Spatial inconsistency and drift estimation tool: The above operations can affect where the virtual object is rendered on Bob's display. We develop a spatial inconsistency and drift estimation tool to quantify these impacts on the rendered virtual objects. This tool runs as an overlay in SPAR, collecting logs from the AR devices, and computing the drift and spatial inconsistency offline. The chief challenge lies in finding a fixed real-world reference point from which to accurate measure how the position of the virtual object changes over time or across users.

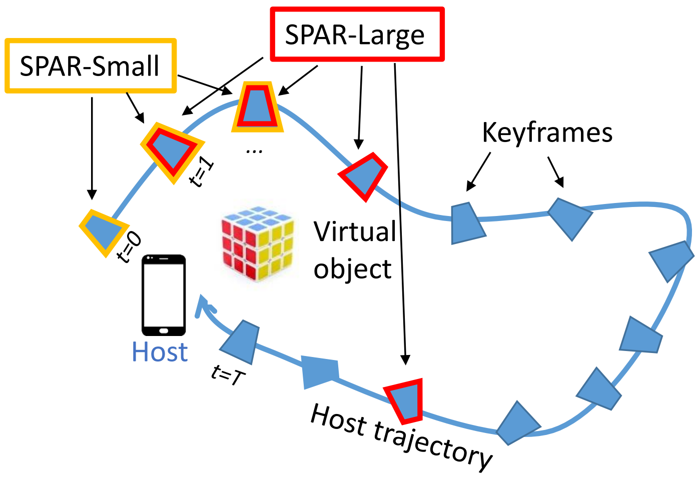


Figure 4.8: Adaptive AR communication strategies. SPAR-Small selects keyframes and point clouds from when the host creates a virtual object. SPAR-Large selects information from when the host is near a virtual object and it is visible.

4.5 System Design

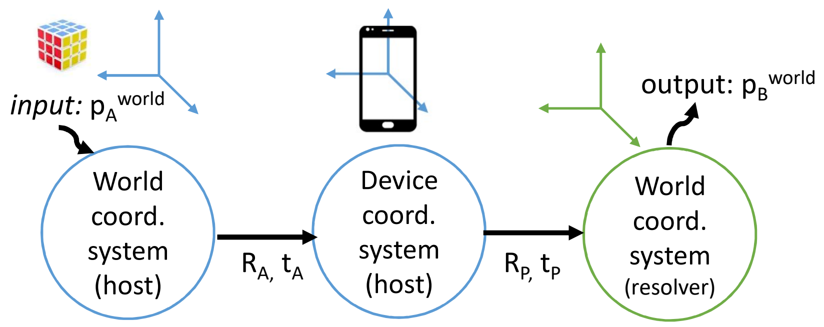
In this section, we discuss the main modules of SPAR: adaptive AR communications (Sec. 4.5.1), coordinate system alignment (Sec. 4.5.2), updated AR rendering (Sec. 4.5.3), and the spatial inconsistency and drift estimation tool (Sec. 4.5.5).

4.5.1 Adaptive AR Communications (Host)

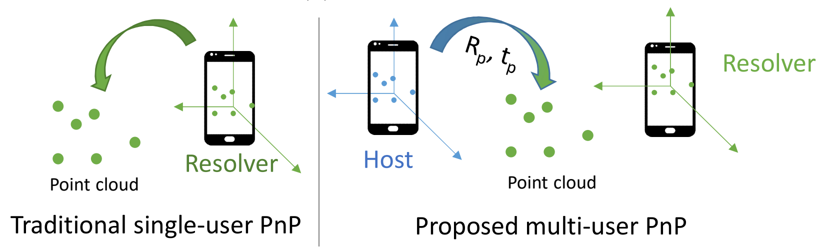
To align the positions of the virtual objects with respect to the real world in each AR display, the host sends a set of keyframes, the associated point cloud with those keyframes, and the virtual objects' coordinates to a resolver. A virtual object's coordinates are a small 4×4 transformation matrix, but set of keyframes can become very large if many keyframes are sent. While a simple strategy is to send all of the keyframes, this can grow very large as the hosting device continues to collect more data, incurring significant communication latency. For example, in our experiments (details in Sec. 4.7), the host collects approximately 110 keyframes after 1-2 minutes, consuming 40 Mb even after applying standard compression techniques [2], resulting in 5 seconds of communication latency on an 8 Mbps uplink connection. Note that these 110 keyframes were already downsampled by SLAM; in other words, the basic downsampling done by SLAM is insufficient and still results in high communication latency. We therefore ask: Can we adapt which keyframes and associated point cloud information are sent, in order to decrease the communication latency?

Our idea is to intelligently adapt what information is sent based on the AR scene, balancing between sending enough information to enable good coordinate system alignment between the host and resolver, but less information to reduce the communication latency. We base our strategies on an insight unique to the AR setting: *only keyframes close to the virtual object in time and space are needed*. We propose two adaptation strategies: (1) an aggressive SPAR-Small strategy (yellow outline in Fig. 4.8), which only selects the keyframes that are close in time to when the virtual object was created, and (2) a more comprehensive SPAR-Large strategy (red outline in Fig. 4.8), which only selects keyframes from when the host is near the virtual object, and the virtual object is visible on the display.

The intuition behind the SPAR-Small strategy is that when the host creates the virtual object by tapping on the device's display, the information about the scene observed around this time is likely sufficient for a resolver to recreate the same virtual objects. The intuition behind the SPAR-Large strategy is two-fold: visibility is important because if the virtual object is not visible on the screen, the camera is usually facing another direction and the observed features may not be useful; and nearness is important because if the host


input: p_A^{world}
World coord. system (host)
R_A, t_A
Device coord. system (host)
R_P, t_P
World coord. system (resolver)
output: p_B^{world}

(a) Overall process.



(b) Multi-user PnP to compute R_p, t_p

Figure 4.9: Coordinate system alignment to transform a virtual object from the host's world coordinate system to the resolver's world coordinate system.

moves in a large environment, the number of keyframes may be large, so nearness can filter out far-away keyframes. The output of this module is a reduced set of keyframes and their associated point clouds, F_A, sent from the host to a resolver.

4.5.2 Coordinate System Alignment (Resolver)

Next, given that a resolver has received the set of keyframes and associated point cloud F_A and the virtual object's coordinates p_A^{world} in the host's world coordinate system, how can a resolver determine the position of the virtual objects in its own world coordinate system, which is needed for rendering? The main challenge is that each device's world coordinate system is initialized at an arbitrary origin point (typically wherever the device is when the AR application is launched), and the devices may be launched randomly by users from different locations in space. Some existing AR apps [37, 73] side step this problem by requiring that the users open their AR apps when the phones are exactly side-by-side, but this is an unnecessary burden on user experience. The goal of the SPAR's coordinate system alignment module is to enable the AR apps to launch from arbitrary starting points, while correctly positioning the virtual objects in each user's AR display.

Multi-user PnP: The specific technical goal is to transform a virtual object's coordinates, which are provided in the host's world cooordinate system (left circle in Fig. 4.9a) to a resolver's world coordinate system (right circle in Fig. 4.9a). An intermediate step is the host's device coordinate system (middle circle in Fig. 4.9a). Since the host knows the definitions of its own world coordinate system and its device coordinate system (left to middle circle in Fig. 4.9a), it can estimate the appropriate transformation R_A, t_A (*i.e.,* its pose) using SLAM (line 5 in Alg. 2). However, the missing transformation is from the host's device coordinate system to a resolver's world coordinate system, R_p, t_p (middle to right circle in Fig. 4.9a).

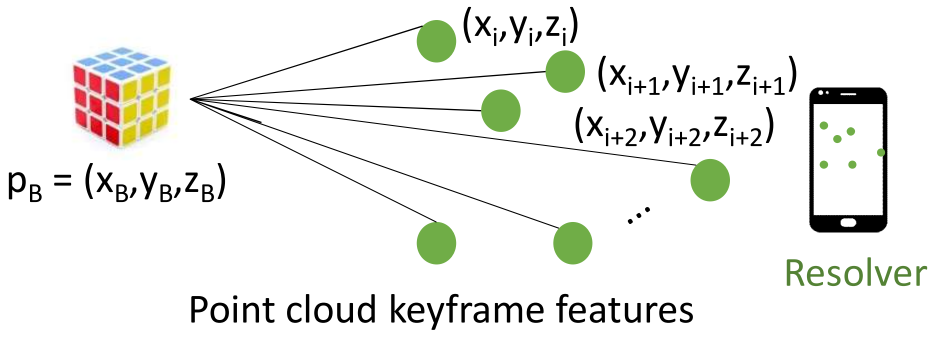
Our key idea is to use real-world features, which remain fixed, to estimate this transformation. Specifically, we propose a multi-user PnP method, which is a re-purposing of the single-user PnP method [99] that already runs on AR platforms. We utilize existing functions in a new way in order to avoid implementation of complex new functionality and keep the system lightweight. However, the existing single-user PnP function can only query for a given keyframe within a device's own world coordinate system, not between two devices' coordinate systems. For example, when running on Bob's device, the inputs to single-user PnP are a query keyframe with associated point cloud, and the output is the an estimate of Bob's pose in Bob's world coordinate system (left side of Fig. 4.9b). However, for multi-user PnP, we realize that we can query one of Alice's keyframes instead, and compute the pose of Alice's device in Bob's coordinate system (right side of Fig. 4.9b), using exactly the same PnP function. This gives us the transformation R_p, t_p (line 6 in Alg. 2), and enables us to compute the complete chain of transformations $R_p(R_A p_A^{\text{world}} + t_A) + t_p$ (line 7 in Alg. 2) needed to render the virtual objects in Bob's display.

Algorithm 2 Coordinate System Alignment
Inputs: set of keyframes and associated point clouds from host F_A, virtual object coordinates in host's world coordinate system p_A^{world}, set of incoming keyframes $\{f_B\}$
Parameters: feature geo distance threshold T_{feature}
Outputs: virtual object coordinates in resolver's world coordinate system p_B^{world}

```
1: $DBoW_A \leftarrow$ ConstructDBoW($F_A$)                          ▷ receive $F_A$ + process
2: for each new keyframe $f_B$ do
3:     $f_A \leftarrow$ MatchKeyframe($DBoW_A, f_B$)
4:     if $f_A \neq$ null then
5:         $(R_A, t_A) \leftarrow$ DevicePose($f_A$)
6:         $(R_p, T_p) \leftarrow$ MultiUserPnP($f_A, f_B$)
7:         $p_B^{\text{world}} \leftarrow R_p(R_A p_A^{\text{world}} + t_A) + t_p$          ▷ coordinate system alignment
8:         if FeatGeoDist($f_B, p_B^{\text{world}}$) $< T_{\text{feature}}$ then return $p_B^{\text{world}}$     ▷ feature geo
   distance filter
```

4.5.3 Updated AR Rendering (Resolver)

Given the initial rendered virtual objects from Sec. 4.5.2, we next discuss what happens as a resolver moves around and collects more information about the real world. Can this new information be used to re-compute the coordinate system alignment and update the poses of the virtual objects for improved spatial consistency? A naive solution of using the most recent keyframe and associated point cloud to update the poses of the virtual objects can lead to bad results, however, if the keyframe was observed at a distant point



(x_i,y_i,z_i)
$(x_{i+1},y_{i+1},z_{i+1})$
$(x_{i+2},y_{i+2},z_{i+2})$
$p_B = (x_B,y_B,z_B)$
...
Resolver
Point cloud keyframe features

Figure 4.10: Feature geo distance is the average distance from real world features to a virtual object.

far from the virtual object. This is because the coordinate systems would be well aligned at that distant point, but lose alignment in areas closer to the virtual objects, resulting in spatial inconsistency with other users. So when should the new information be used?

Our approach is to develop a new metric to help estimate which new keyframes and associated point clouds will produce the best alignment and improve the spatial consistency. Simple estimation schemes such as selecting keyframes from the host and resolver with the lowest pixel-by-pixel differences or Hamming distance are possible, but in our experience, this results in keyframe matches that are distant from the virtual object, resulting in poor coordinate system alignment and thus spatial inconsistency issues.

Instead, we leverage our understanding of the AR scenario: *features near the virtual object are more important for coordinate system alignment.* We propose a new metric based on this, "feature geo distance". The feature geo distance is defined as the average distance to a virtual object over the common features in a pair of matched keyframes, as illustrated in Fig. 4.10. Specifically, given a set of features i with coordinates (x_i, y_i, z_i) from keyframe f_B, and the 3D coordinates of the virtual object p_B^{world}, we compute the feature geo distance as:

$$\texttt{FeatGeoDist}(f_B, p_B^{\text{world}}) = \frac{\sum_{i \in f_B} ||(x_i, y_i, z_i) - p_B^{\text{world}}||}{M}$$

where M is the number of common features. We compute the feature geo distance on each pair of matched frames, and only update the pose of a virtual object if the feature geo distance is less than a threshold (line 8 in Alg. 2). Essentially, we use feature geo distance as a predictor of spatial inconsistency (this correlation is evaluated in Sec. 4.7.3).

4.5.4 Motivation of measurement tool: Issues with Manual Labeling and Absolute Trajectory Error

In this section, we provide insight into why manual labeling or ATE are insufficient for measuring drift of an AR virtual object. Note that neither manual labeling nor ATE measurements are needed for casual users of SPAR, but are only needed to evaluate SPAR in this chapter.

Manual Labeling: In the case of manual labeling, in our experience, it took approximately 10-30 seconds to hand-annotate the position of a virtual object in each frame. For an AR app updating its display at 30 frames per second (FPS) running for 5 minutes, this could take up to 1.25 hours to measure a virtual object's drift for the duration of a user's experience. Clearly, this is infeasible and unwieldy, particularly if multiple users are participating in the AR experience and each of their frames need to be annotated.

Absolute Trajectory Error: ATE does not provide sufficient information about the position of the *virtual object*, since the device trajectory and ATE only have information about the position of the *device.* The device position is insufficient knowledge about the virtual object, because rendering the virtual object involves projecting the virtual object onto the AR display, which requires both device position and rotation provided by SLAM. Therefore, ATE alone cannot tell the AR device where the virtual object is rendered on the display, and hence what its spatial drift is.

We next describe an experiment we conducted to illustrate why ATE cannot be used to evaluate the spatial drift of an AR virtual object; *i.e.,* why ATE or the device trajectory does not accurately capture the spatial drift of an AR virtual object. This experiment is illustrated in Fig. 4.11. We use ARCore as the AR platform. We started the experiment by creating a virtual object (the tower of shapes) on the floor, and then move backward (in the y direction), without any left/right movement (in the x direction). The height of the device is also fixed so there is also no up and down movement (z direction). In the 3D plot of Fig. 4.11, we plot the SLAM-estimated device trajectory (blue line), which is a straight line in the XY plane. The SLAM-estimated trajectory matches well with the ground truth device trajectory (red line). Intuitively, we might expect this accurate device position estimate to mean the virtual object won't drift.

However, the SLAM-estimated device trajectory gives us no information about the position of the virtual object on the display, and the virtual object does in fact drift, despite the accurate device position estimates. In Fig. 4.11, we show screenshots of the virtual object at 3 different times. At time 1, the virtual object is directly above the piece of paper. At time 2, the virtual object drifts backward from the paper (towards the user), and at time 3 it drifts forward (away from the user). We don't know where the virtual

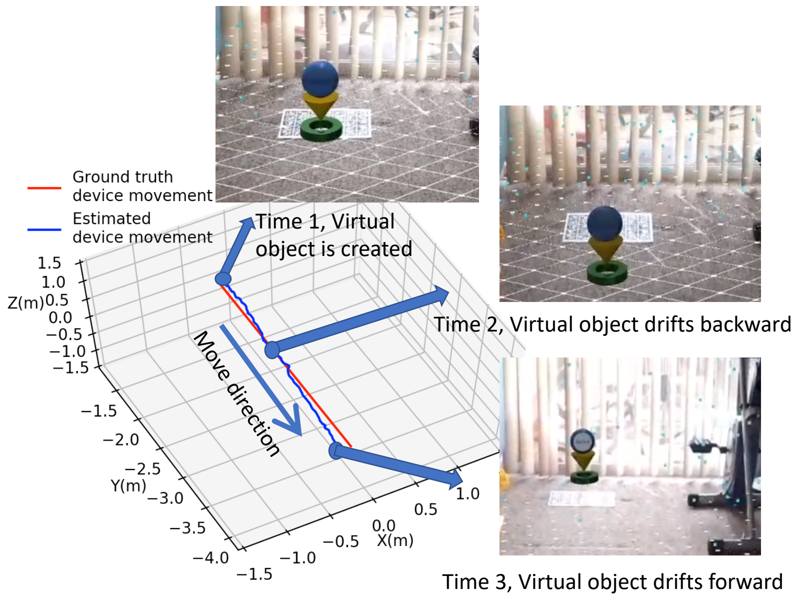


Figure 4.11: The virtual object drifts back and forth as the user moves. However, simply looking at the device trajectory/ATE alone gives little information about how the virtual object drifts.

object is or how much the virtual object drifts by looking at the device trajectory alone, until we see the screenshots of the virtual object. In other words, the accuracy of the device trajectory estimation is not tightly correlated with the drift of the virtual object.

Moreover, in the multi-user scenario, each AR app can update its virtual object position, and then our tool can compute the position difference (spatial inconsistency) of the virtual object between any two users. However, we can't compute the spatial inconsistency across multiple users just from ATE or the trajectory because of the lack of time synchronization and rotation information from ATE alone.

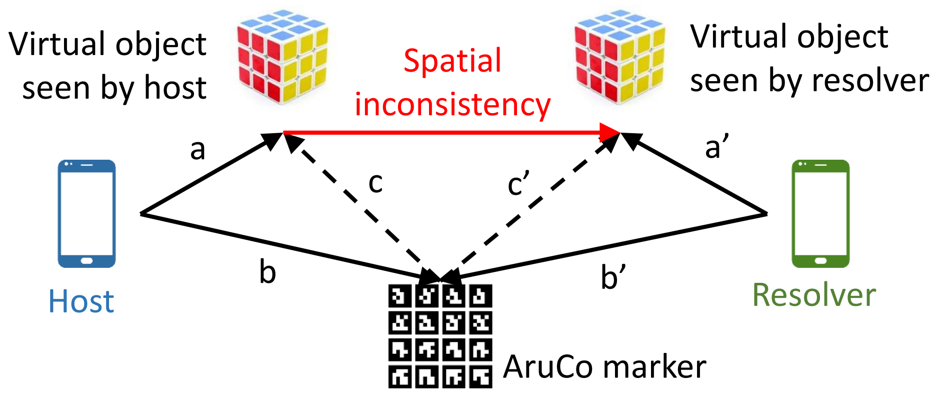


Figure 4.12: Conceptual example of computing spatial drift and inconsistency.

4.5.5 Design of Spatial Inconsistency and Drift Estimation Tool

In this section, we discuss our design of a tool to automate and improve human eyeball-based accuracy estimation.

Overall idea: Our method, in short, is to place one or more markers (*e.g.*, ArUco markers [84]) into the real world before running the AR app. The markers' location and orientation can be accurately estimated by the devices using PnP [99, 6], and used as reference points from which to measure and compare the virtual object's position as rendered by each device. We use ArUco markers as opposed to natural features in the scene because pose estimation from natural features is not accurate enough for our purposes, resulting in extremely noisy tool output.

As shown in Fig. 4.12, Alice and Bob will draw the virtual object at a and a', respectively, relative to their current pose. In an ideal multi-user AR scenario, the virtual object's resulting locations (end of a and a' arrows) are exactly the same; however, in practice they can be spatially inconsistent (red line in Fig. 4.12). In order to measure the difference between the endpoints of a and a' (*i.e.,* the spatial inconsistency), the marker is used as a common reference point. Specifically, we develop the following methodology:

1. Using the keyframe f and f' from the host and a resolver, respectively, the tool calculates the b and b' vectors from the device to the marker (see Fig.4.12) using the marker present in the keyframe and the single-user PnP method. (lines 1-2 in Alg. 3)

2. The c and c' vectors from the real-world marker to the virtual anchor are computed as $c = a - b$, $c' = a' - b'$. (lines 3-4 in Alg. 3).

3. The tool outputs $||c - c'||$ (length of the red line in Fig. 4.12), which is the magnitude of the spatial inconsistency between the two devices. (line 5 in Alg. 3).

The above procedure estimates the spatial inconsistency of multiple users at a single instance in time. A similar procedure can be used to compute the spatial drift of a single user over time, by letting a, b, c be Alice's observations at $t = 0$, and a', b', c' be Alice's observations at $t = 1$. The output $||c - c'||$ in that case represents the spatial drift between $t = 0$ and $t = 1$. An additional wrinkle is that the above steps rely on a, b, c being in the same coordinate system. However, because they are computed by different functions in the AR platform, they typically are not output in the same coordinate system, because of the way that the real world and the devices are modeled and stored in AR/VR systems [59, 86]. We use a variation of the techniques described in Sec. 4.5.2 to reconcile these coordinate systems and produce the spatial inconsistency and drift estimates.

Algorithm 3 Spatial Inconsistency Estimation

Inputs (device 1): virtual object coordinates a, coordinate system transformation g_0, frame f
Inputs (device 2): virtual object coordinates a', coordinate system transformation g'_0, frame f'
Outputs: Spatial inconsistency $||c - c'||$

```
1: $(g_2, b) \leftarrow$ SingleUserPnP($f$)                    ▷ device 1 estimate
2: $(g'_2, b') \leftarrow$ SingleUserPnP($f'$)                 ▷ device 2 estimate
3: $c \leftarrow g_1(g_0(a)) - g_2(b)$
4: $c' \leftarrow g_1(g'_0(a')) - g_2(b')$
5: return $||c - c'||$                                ▷ output spatial inconsistency
```

4.6 Implementation

AR devices: We use Samsung Galaxy S7 smartphones running Android 8 as the AR devices. Our AR system is developed in C++ with the Android NDK on top of VINS [68, 76, 60], an open-source AR platform, which we extend with multi-user capabilities with 3000 additional lines of code. VINS has comparable baseline spatial drift issues as ARCore, as shown in Fig. 4.2b, so we believe it is a reasonable platform on which to test our proposed systems. We use FAST feature and BRIEF descriptors. For coordinate system alignment, we use DBoW2 [29] to speed up keyframe matching, OpenCV [16] for basic PnP functionality, and Boost [2] for data serialization and compression for efficient network transmissions. Communications are accomplished through sockets, where the host acts as a socket server and the resolvers connect to download the set of keyframes via a TCP connection. The devices are connected to a TP-Link AC1900 WiFi router.

Replay framework: To enable experimentation with different algorithms under repeatable conditions (*e.g.*, the same mobility patterns of the users), we developed a replay framework that replays the same sequence of camera frames from the host and a resolver, and allows SPAR or other baselines to be run on top. This involves saving the host's point cloud, keyframes, and virtual objects' coordinates, along with the resolvers' point clouds and keyframes. Then for each trial, the framework loads the host's point cloud and keyframes, run the desired algorithms, and emulates a resolver's experience as new camera frames arrive one-by-one, adaptive communications occur, and coordinate system alignment and updated AR rendering are performed. If successful, the virtual object will be drawn on the resolver's keyframes and we record the latency and spatial inconsistency.

Tool: The spatial drift and inconsistency tool runs offline on an edge server in our implementation; in general, it could be run on any device, including one of the AR devices themselves. It is written in Python3 using OpenCV [17], Numpy, and its quaternion library. The tool takes as input the keyframes from each AR device, the AR app's log of the corresponding virtual object positions and orientations, the measurements of the ArUco marker, and the calibration matrices of AR device cameras. Since that different AR platforms (*e.g.*, VINS, ARCore) typically output data in slightly different formats, we wrote ad hoc parsing and conversion functions for VINS.

4.7 Evaluation

SPARWe evaluate SPAR's performance in terms of computation and communication latency, bandwidth consumption and spatial inconsistency. The main findings are that SPAR-Small and SPAR-Large lower average latency by up to 55% on average, and provide improved spatial inconsistency, especially when a virtual object first appears, by 11%-60%, compared to baseline approaches.

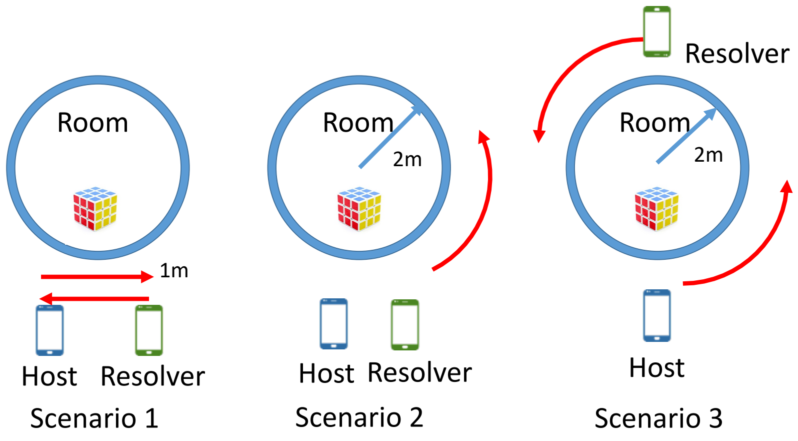


Figure 4.13: User mobility pattern test cases.

4.7.1 Setup

Scenarios: We perform experiments in the lab and in a home environment. A host places a virtual object (in our case, a virtual cube), walks around the area, and sends the relevant data once to a resolver. This resolver then walks around the scene and tries to render the virtual cube at the correct location. We evaluate the performance from the point of view of a resolver at two time instances: (i) initially, when the virtual object is first displayed (Sec. 4.7.2), and (ii) subsequently as the resolver continues to move around and update the virtual object's position (Sec. 4.7.3). We also evaluate the performance of the tool to estimate spatial drift and inconsistency during these two phases (Sec. 4.7.4). The average WiFi speed was 8 Mbps upload and 50 Mbps download in the lab, and 8 Mbps upload and 20 Mbps download at home.

The user mobility pattern test cases are illustrated in Fig. 4.13 and described below.

- *Scenario 1: Small area:* The host and a resolver are mostly stationary, and move within a 1 m $\times$ 1 m area.
- *Scenario 2: Large area + same initial position:* The host and a resolver start at the same place facing the same direction. Each user moves independently within a 4 m $\times$ 4 m area.
- *Scenario 3: Large area + different initial position*: The host and resolver start at different places. They both move independently within a 4 m $\times$ 4 m area.

Each scenario is repeated 25 times, with each trial lasting 40-90 seconds.

Baselines: Along with the SPAR-Large and SPAR-Small strategies proposed in Sec. 4.5.1, we compare against several baselines, All and ARCore:

- *All:* The host sends all keyframes and associated point clouds (already downsampled by SLAM) to a resolver.
- *SPAR-Small:* The host sends 5 keyframes and their associated point clouds from before and after creating a virtual object (10 time instances total). This strategy geared towards small environments.
- *SPAR-Large:* The host sends keyframes and associated point clouds for which the virtual object is visible within the FoV and the host is within $T_{\text{keyframe}} = 3$ m of the virtual object. This strategy is more conservative and geared towards large environments.
- *ARCore [32]:* ARCore is a highly optimized, closed-source production level AR platform with cloud processing. We include this comparison for reference; our goal is to showcase the improvements of our proposed methods in the open-source VINS platform, which could then be incorporated into optimized production platforms. Due to API restrictions,

the ARCore experiments differ slightly in that VINS creates a virtual object before the host moves, while ARCore creates it after the host moves.

We also experimented with two other baselines using the individual nearness and visibility criteria from SPAR-Large, but their results were similar to the other baselines and omitted. We didn't compare performance with other AR platforms such as Hololens or ARKit because they run on different hardware (Hololens, iPhone/iPad), so it is difficult to have a fair comparison with SPAR, which is prototyped on Android (although its methods are generalizable).

Metrics: We evaluate several metrics:

- *Latency of Initial Virtual Object Appearance (s):* This latency consists of several components:
  - *Save*: The time the host spends to adapt the AR data in preparation for transmission.
  - *Communication*: The time spent to communicate the selected data to a resolver. For ARCore, this includes the cloud processing time.
  - *Load*: The time a resolver spends to load the host's data and initialize SLAM processing.
  - *Resolve*: The time for a resolver to move close to a virtual object and perform coordinate system alignment.
- *Spatial drift and inconsistency (cm):* As discussed in Sec. 4.2, spatial drift is defined as the distance that a resolver's virtual object changes over time(assuming a ground truth

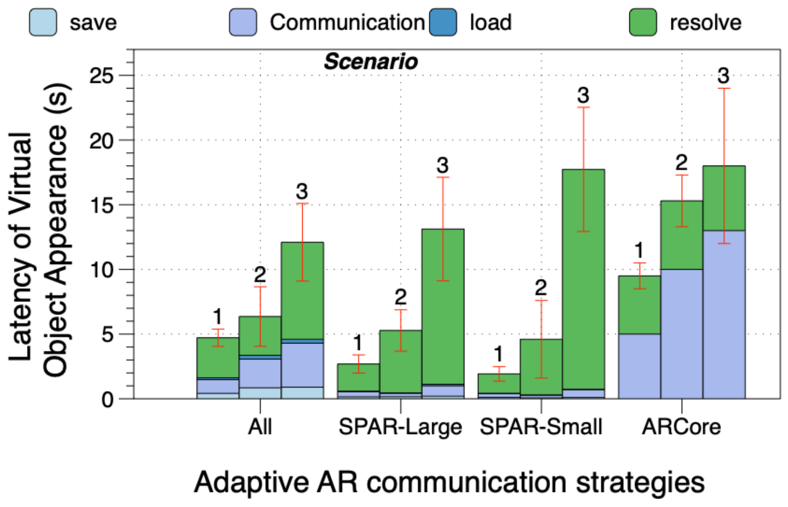


Figure 4.14: SPAR reduces total latency by up to 55% compared to All and ARCore baselines, on average. Note that the latency here is the initialization latency, when the user first loads the AR app. Once this initialization has happened, subsequent updates to the virtual objects' locations and orientations happen in real-time.

stationary virtual object). Spatial inconsistency is defined as the distance between a host and resolver's virtual object instances at a given time.

- *Failure rate:* A virtual object can fail to appear on a resolver's screen if the coordinate system alignment cannot find similar enough matching frames. We count the number of times this failure occurs.

4.7.2 Initial AR rendering

We first discuss the initial rendering of the virtual object on the resolving user's display. We seek to answer the following questions: Does the adaptive AR communication strategy reduce latency? Are the virtual objects rendered with low spatial inconsistency?

Latency: We plot the average latency of a virtual object's initial appearance, along with its breakdown, in Fig. 4.14 for each scenario and baseline method. The SPAR-

Large and SPAR-Small strategies generally have lower latencies than the All and ARCore baselines, with SPAR-Small performing well in small environments like scenario 1, and SPAR-Large generally performing well in larger environments such as scenarios 2 and 3. The All baseline generally has higher latency than SPAR because it sends the full AR data, resulting in more than 3 seconds of communication latency. One exception is scenario 3, where SPAR takes more time than All. This is because the latency measurement includes the time for the user to walk closer to the virtual object. SPAR uploads fewer keyframes and typically needs more time to find a keyframe match in scenario 3, but once the virtual object does appear, it has significantly lower spatial inconsistency, as discussed later on in Fig. 4.16a. This illustrates the tradeoff between latency and spatial inconsistency. We also note that scenario 3 is considered a challenging scenario, with many off-the-shelf AR apps (such as Pokemon Go and Just a Line) simply avoiding such scenarios by asking players to stand side-by-side during initialization.

Finally, the ARCore baseline also has high total latency, because it sends large amounts of data for cloud processing. In general, there is a tradeoff between the communication and resolve latency: a low communication latency (as in SPAR-Small) implies scanty information for coordinate system alignment, so a resolver has to take more time to find a match before coordinate system alignment is successful, resulting in higher resolve latency.

In summary, the SPAR-Large strategy achieves good balance of communication and resolve latency, and can save an average of 15% total latency compared to All and 40% compared to ARCore, on average across all scenarios.

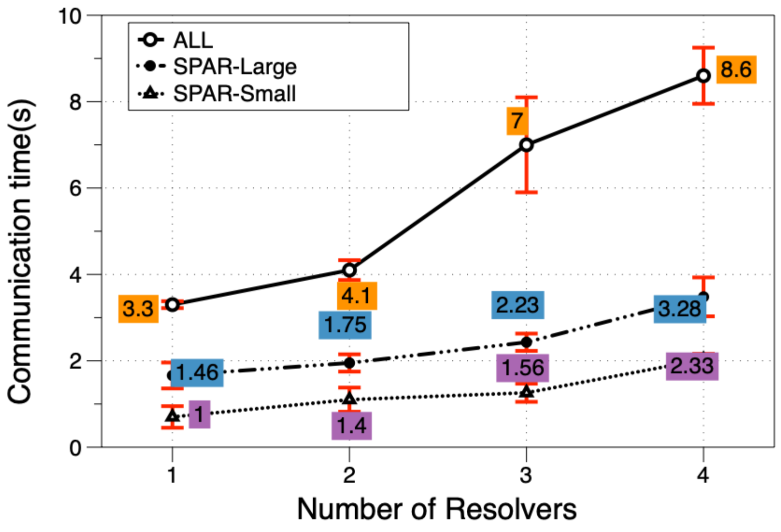
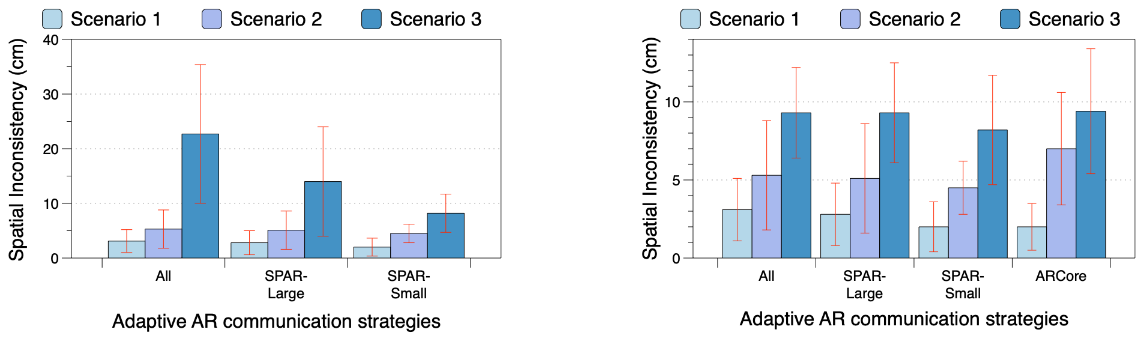


Figure 4.15: SPAR scales communication time with the number of resolvers.



(a) First appearance of virtual object.

(b) Resolver 1m away from virtual object.

Figure 4.16: SPAR improves spatial inconsistency, especially in large areas (scenarios 2, 3) by 11%-60% on average, compared to All and ARCore.

Scalability: We also examine how SPAR scales as the number of resolvers increases, by varying the number of resolvers from 1 to 4. Since all of the resolvers communicate with the host simultaneously over a shared bottleneck wireless connection, we focus on the communication latency only, as the save, load and resolve processes run in parallel on the individual devices and thus scale up easily. The average communication latency across all scenarios is shown in Fig. 4.15. Both SPAR-Small and SPAR-Large scale well with the number of resolvers, with approximately 0.5 s of latency for each additional resolver. However, All suffers from long communication latency when there are more than 2 resolvers.

Spatial inconsistency: We next examine the virtual objects' spatial inconsistencies, and plot their mean and standard deviation at two time instances: when a virtual object initially appears on a resolver's display, typically far away (Fig. 4.16a), and later when a resolver moves closer, around 1 m from a virtual object (Fig. 4.16b). The reason we plot two different time instances is because as a resolver moves closer to a virtual object, it observes more information about the environment and can update the position of the virtual object, changing the spatial inconsistency values.

SPAR-Small performs well at the initial appearance of the virtual object (< 8 cm spatial inconsistency in all scenarios), and reduces the spatial inconsistency as the resolver gets closer to the virtual object. One drawback of SPAR-Small is that it has high latency in the large environments it was not designed for (see Fig. 4.14). In large environments (scenarios 2 and 3), SPAR-Large has lower spatial inconsistency than the All baseline when the virtual object first appears (Fig. 4.16a), and compared to the ARCore baseline when close to a virtual object (Fig. 4.16b). Hence SPAR-Small and SPAR-Large work well for the respective environments they were designed for. Examples from scenario 2 are shown in Fig. 4.17.

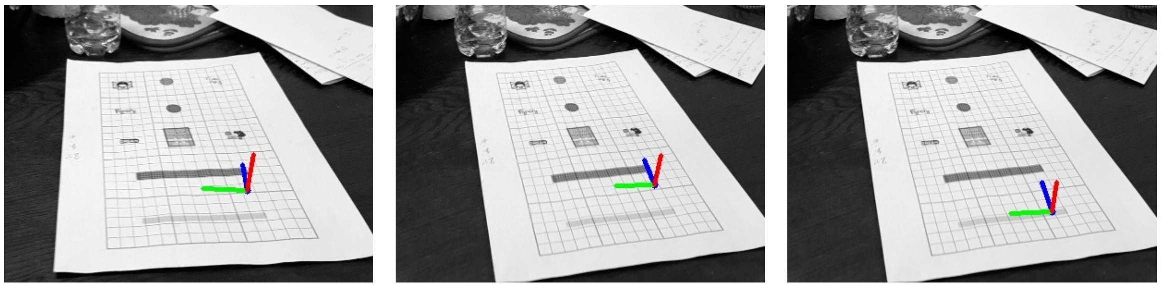
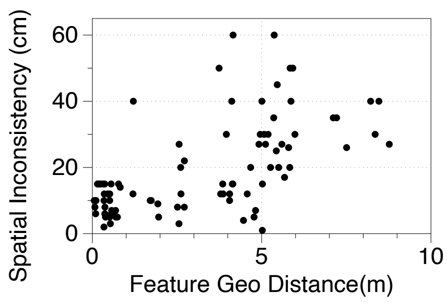

(a) Host (b) SPAR-Small (c) SPAR-Large

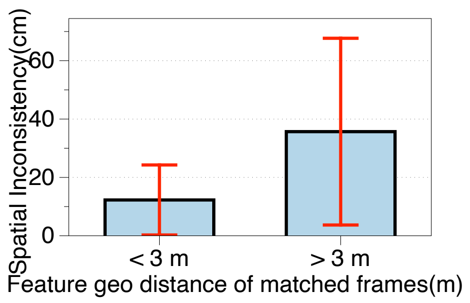
Figure 4.17: Screenshots of the virtual object seen by the resolver under different adaptive AR communication strategies.

Surprisingly, the All baseline does not have the lowest spatial inconsistency, despite communicating full information about the environment. This is because the abundance of information sometimes results in coordinate system alignment far from the virtual object, leading to poor alignment near the virtual object and thus spatial inconsistencies. ARCore performs worse in the larger scenarios 2 and 3 when a resolver is close to the virtual object (Fig. 4.16b). Note that we do not record ARCore's initial spatial inconsistency because the resolver is too far away from the virtual object to measure clearly (SPAR does not have this issue because it can produce detailed logs for analysis).

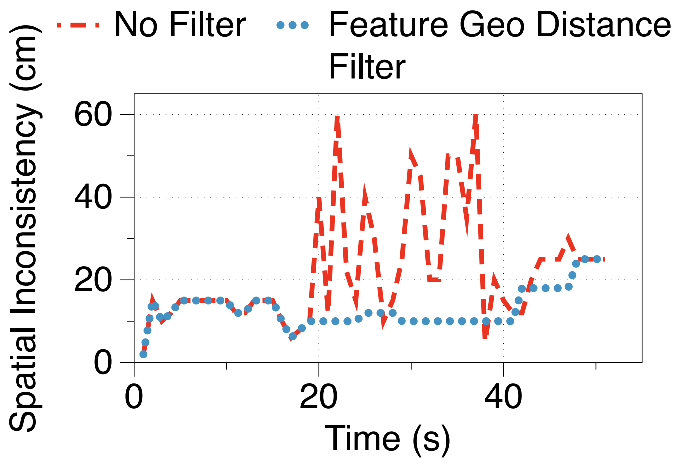
In summary, SPAR-Small's spatial inconsistency in small scenarios ranges from 2-3 cm at a virtual object's first appearance, which is 20% better than the All baseline; while SPAR-Large achieves 6-9 cm spatial inconsistency in large scenarios, which is 11%-35% better than ARCore when near a virtual object. SPAR's accuracy in scenario 1 and 2



(a) Correlation.



(b) T_{feature} threshold.



(c) Time series.

Figure 4.18: The feature geo distance metric filters good keyframes for coordinate system alignment, resulting in lower spatial inconsistency for a resolver.

is generally consistent with or improves over ARCore, with most challenging scenario being scenario 3, where SPAR still outperforms ARCore on average.

Failure rates: In our experiments, SPAR-Small failed to resolve twice in scenario 2. Since we have 75 trials total across scenarios, this gives a failure rate of 2.7%. The cause of failure may be because SPAR-Small too aggressively reduces the amount of AR data transmitted, as it only save 10 keyframes and their associated point cloud, making it hard to perform coordinate system alignment and render a virtual object. The other baselines did not fail throughout our experiments, so on the whole, despite SPAR-Small having lower spatial inconsistency and good latency, SPAR-Large is preferable in general for its more consistent performance.

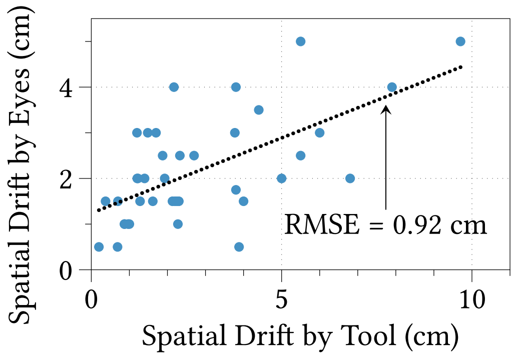
4.7.3 Subsequent AR rendering

In this section, we isolate the impact of SPAR's "Updated AR Rendering" module (Sec. 4.5.3). To validate our hypothesis that feature geo distance correlates with spatial inconsistency (see Sec. 4.5.3), we plot the spatial inconsistency versus feature geo distance in Fig. 4.18a over 3 trials. Each point on Fig. 4.18a represents a specific pair of matched keyframes; the y-axis records the spatial inconsistency resulting from coordinate system alignment with that matched pair. We can see that as the feature geo distance increases, spatial inconsistency gets worse. This suggests that feature geo distance can be used to select good keyframes for coordinate system alignment, and thus improves the virtual object's spatial inconsistency.

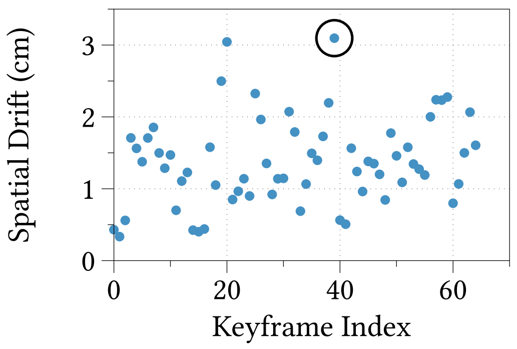
Since we use a feature geo distance threshold $T_{\text{feature}} = 3$ m in Alg. 2, in Fig. 4.18b we plot the average spatial inconsistency and standard deviation when the feature geo distance is less than and greater than the threshold. It includes 6 trials and 310 matched keyframes, with 170 frames having geo distance less than 3 m, and 140 frames greater than 3 m. The average spatial inconsistency for frames with feature geo distance greater than 3 m is nearly 40 cm, but applying the threshold filters out those frames and reduces spatial inconsistency by more than 50%. This reinforces our message that the feature geo distance can be an efficient way to filter out keyframe matches that result in larger spatial inconsistency.

Finally, to illustrate how the feature geo distance metric impacts AR rendering, in Fig. 4.18c we plot the time series of a particular trial in scenario 2. We compared our "feature geo distance filter" approach (blue line) to a simple "no filter" baseline (red line) that updates a virtual object's position using the resolver's most recent keyframe. Since in scenario 2, a resolver is initially near the virtual object, then moves away, then moves close again, the expectation is that the feature geo distance of the most recent keyframe will follow a similar pattern, first being low, then high, then low, Because the baseline approach uses the most recent keyframe for matching, this suggests that the virtual object's spatial drift will get worse and then better. Fig. 4.18c shows the baseline approach matches our expectation, while our proposed approach achieves a better (lower) spatial drift by intelligently selecting the right keyframes according to the feature geo distance metric.

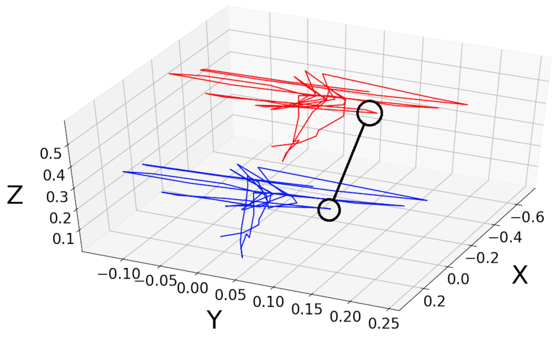
In summary, the feature geo distance metric provides a good way to select which keyframe the resolver should use to update the virtual object's position, and can reduce spatial drift by 50% on average compared to a baseline "no filter" approach of using the most recent keyframe for coordinate system alignment.



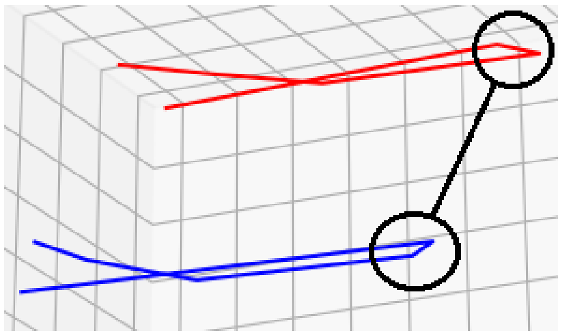
(a) Manual vs. automatic labeling.



(b) Spatial drift over time.



(c) Trajectory of the device.



(d) Zoomed in of (c).

Figure 4.19: Spatial drift and inconsistency estimation tool. The tool matches manual human labeling with an RMSE of 0.92 cm.

4.7.4 Spatial Drift and Inconsistency Tool

In this section, we evaluate the final component of SPAR, the spatial drift and inconsistency tool proposed in Sec. 4.5.5. We wish to compare the drift/inconsistency values reported by the tool vs. the human-observed values, in order to evaluate the tool's accuracy. We first evaluate the tool's performance qualitatively. We plot an example time series of the tool's output in Fig. 4.19b. This time series shows that the virtual object moves by less than 3 cm every 1 second or so, which we qualitatively observe to be true during the experiment. To understand these results, in Fig. 4.19c we plot the trajectory of the resolver in space, with respect to the virtual object (blue line, a from Fig. 4.12) and

ArUco marker (red line, b from Fig. 4.12). These trajectories are identical, except for an offset, as expected since they are with respect to different reference points. However, it is when this offset changes over time ($c = a - b$) that spatial drift occurs. We can see this in Fig. 4.19b and Fig. 4.19d, where the circled points correspond to varying offset and thus higher spatial drift.

To evaluate the tool's performance quantitatively, we prepare the following test setup. We place a real 1 cm $\times$ 1 cm grid paper in the scene, initialize the virtual object on top of the grid paper, and painstakingly go through each keyframe and manually record the coordinates of the virtual object on the grid paper. We then choose random pairs of keyframes and plot the spatial drift from the manual labeling vs. the spatial drift output by the tool. Fig. 4.19a shows the results. The RMSE is 0.92 cm. We see good agreement between the manual labels and the tool's output, indicating that our proposed method can successfully estimate spatial drift (spatial inconsistency is computed in a similar manner). Any disagreement between the manual labels and the tool's output are, we believe, due to fundamental limitations of SLAM in computing the device trajectory (*e.g.*, Fig. 4.19c), which the tool relies on. In terms of computation time, the tool is able to generate estimates for tens of keyframe pairs in about one second, whereas manual labeling by humans takes several seconds per keyframe pair.

4.8 Discussion

Multiple virtual objects: Although our experiments focused on sharing one virtual object between users, SPAR could easily generalize to multiple virtual objects, because the common coordinate system it computes (Sec. 4.5.2) can be used to represent the poses of multiple virtual objects. Specifically, for each resolver, coordinate system alignment would be performed once, and each virtual object projected and rendered onto the AR display based on its pose in the computed coordinate system. This would result in the each virtual object experiencing similar spatial inconsistency and latency as the single object case.

Scalability: The spatial inconsistency experienced by SPAR users would not be substantially impacted as number of users increases. This is because each resolver performs its computations (Sec. 4.5.2, Sec. 4.5.3) in parallel with other clients, so the computed coordinate system, virtual object poses, and hence spatial inaccuracy results of each resolver are independent of each other. This is similar to performing a 2-user (a host and a resolver) experiment multiple times. The main performance bottleneck that depends on the number of users is the communication bandwidth, which impacts the latency, as shown in Fig. 4.15 and discussed in Sec. 4.7.2.

SLAM and marker-based AR: SPAR is designed for SLAM-based AR, which is common in off-the-shelf AR systems such as Google ARCore, Apple ARKit, and Microsoft Hololens. We use VINS-MONO [60] as the basis for our AR system, which is designed for static environments, so SPAR inherits these limitations (SLAM in dynamic environments is an active area of research). Another class of AR systems is marker-based AR [31]. Since marker-based AR only need the marker information to position and render the virtual objects, the host only needs to distribute the marker information (*e.g.*, ArUco marker), rather than keyframes and features as in SLAM-based AR. The marker information can be compactly represented as an image or ID number, and thus is communication-efficient in which case SPAR is not needed.

Cloud vs. P2P architectures: SPAR currently uses a P2P architecture to distribute AR information directly to each resolving client, as do Apple ARKit and Microsoft Hololens. A P2P architecture is a natural fit for AR, since AR information only needs to be distributed in a geographically restricted area. However, SPAR could be modified to run coordinate system synchronization on a central node, such as a cloud or edge server (for example, Google ARCore uses the cloud), although privacy is a concern. In this case, communication latency may increase, but computation latency may decrease, requiring further evaluation of the tradeoffs.

4.9 Related Work

Mobile AR systems: Object detection and image recognition for AR, on device or offloaded to the edge/cloud, has been investigated [20, 39, 61, 80, 62, 96, 11, 52, 64] in order to place virtual objects in the real world. These works are orthogonal to ours as we assume that the virtual objects' locations is given (by object detection or user input), and we focus on how AR users can coordinate this information with others. VisualPrint [53] uses visual fingerprints for localization, whereas we use SLAM for localization as common in commercial AR platforms. While MARVEL [19] studies 6-DoF based AR systems, they assume the real world is pre-mapped, whereas we assume that devices are placed in an unknown environment. Edge-SLAM [10] considers offloading parts of SLAM to an edge server, whereas SPAR does not require infrastructure support. GLEAM [75] focuses on lighting rendering for virtual objects, which is complementary to this work. Recent work [91] proposes geo-visual techniques for fast localization in urban areas; however, their AR system is single-user whereas we focus on multi-user scenarios.

Multi-user AR: CARS [97] shares results from object detection among multiple users, whereas this chapter focuses on more general 3D coordinate system alignment to share virtual objects including those placed by object detection. CarMap [8] proposes efficient map compression, without any virtual objects; in contrast, SPAR uses knowledge of the virtual object positions when deciding what to communicate. Several works [81, 12] present only preliminary measurements of multi-user AR. While industry multi-user AR systems such as Google ARCore [35], Apple ARKit [14], and Microsoft HoloLens [67] are close-sourced, we study communication and spacial inconsistency aspects of multi-user AR through an open-source system [60].

Multi-agent SLAM: Some SLAM systems [44, 7] focus on coordinate system alignment, while other work [51, 24] assumes advanced sensors such as 2D laser scanner or 3D LiDARs. In contrast, this chapter focuses on efficient SLAM-based communications on commodity smartphones, which have a large potential user base. Zou et al. [103] hardcodes transmitting the SLAM data up to every 5 frames, while CCM-SLAM [87] transmits SLAM information whenever it is updated. Instead, we select the appropriate keyframes and their associated point clouds based on the locations of the virtual objects. This is done on top of the default keyframe selection already performed by SLAM frameworks such as ORB-SLAM2 [70] or VINS [76].

In terms of frameworks, we work with VINS-AR [60], which is an Android version of VINS-Mono [76], both of which are single-user SLAM and do not consider communication and consistency issues of multi-user AR. Other open-source SLAM systems are either not tested on Android [87, 28] or do not utilize IMU sensors [70].

4.10 Summary

In this chapter, we investigated communication and computation bottlenecks of multi-user AR applications. We found that off-the-shelf AR apps suffer from high communication latency and inconsistent placement of the virtual objects across users and across time. We proposed solutions for efficient data communications between AR users to reduce latency while maintaining accurate positioning of the virtual objects, as well as a quantitative method of estimating these positioning changes.

Our implementation on an open-source Android AR platform demonstrated the efficacy of the proposed solutions. Future work includes extending our spatial inconsistency tool to other AR platforms such as ARCore, as well as incorporating depth cameras.

Chapter 5

Conclusions

Augmented reality algorithms require the understanding of the environment and can be categorized into two major types: deep learning based AR and SLAM based AR. In this work, we talked about three systems: DeepDecision, change detector and SPAR. These systems study the tradeoffs between accuracy and latency of augmented reality applications where DeepDecision and change detector target deep learning based AR and SPAR targets SLAM based AR.

DeepDecision is a measurement-driven mathematical framework that effectively schedules the deep learning to be executed either locally on mobile devices or offloaded to the edge server. DeepDecision sets the resolution and quality of input video based on latency and accuracy requirements and network conditions. Change detector is a lightweight machine learning algorithm that detects significant changes from the input video so that deep learning will only be applied to significant different frames to save energy and reach real-time performance without offloading. SPAR is a SLAM-based, multi-user mobile augmented reality application that has efficient data communications between users to reduce latency while maintaining accuracy of the virtual objects. SPAR also has an automatic measurement tool to evaluate the drift of virtual objects for both single-user and multi-user scenarios.

Bibliography

[1] Artoolkit. `http://www.hitl.washington.edu/artoolkit/`.

[2] Boost c libraries. `https://www.boost.org/`.

[3] libstreaming. `https://github.com/fyhertz/libstreaming`, 2017.

[4] Tensorflow android camera demo. `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android`, 2017.

[5] xiph.org video test media. `https://media.xiph.org/video/derf/`, 2017.

[6] D. F. Abawi, J. Bienwald, and R. Dorner. Accuracy in optical tracking with fiducial markers: an accuracy function for artoolkit. In *IEEE ISMAR*, Nov 2004.

[7] Mahmoud A Abdulgalil, Mahmoud M Nasr, Mohamed H Elalfy, Alaa Khamis, and Fakhri Karray. Multi-robot slam: An overview and quantitative evaluation of mrgs ros framework for mr-slam. In *International Conference on Robot Intelligence Technology and Applications*, pages 165–183. Springer, 2017.

[8] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. Carmap: Fast 3d feature map updates for automobiles. In *USENIX NSDI*, pages 1063–1081, 2020.

[9] D Stalin Alex and Amitabh Wahi. Bsfd: Background subtraction frame difference algorithm for moving object detection and extraction. *Journal of Theoretical & Applied Information Technology*, 60(3), 2014.

[10] Ali J Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. Edge-slam: edge-assisted visual simultaneous localization and mapping. In *ACM MobiSys*, pages 325–337, 2020.

[11] K Apicharttrisorn, X Ran, J Chen, SV Krishnamurthy, and AK Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. *ACM SenSys*, 2019.

[12] Kittipat Apicharttrisorn, Bharath Balasubramanian, Jiasi Chen, Rajarajan Sivaraj, Yi-Zhen Tsai, Rittwik Jana, Srikanth Krishnamurthy, Tuyen Tran, and Yu Zhou. Characterization of multi-user augmented reality over cellular networks. In *IEEE SECON*, 2020.

[13] Apple. Arkit - apple developer. `https://developer.apple.com/arkit/`.

[14] Apple. Creating a multiuser ar experience. `https://developer.apple.com/documentation/arkit/creating_a_multiuser_ar_experience`.

[15] Apple. Swiftshot. `https://developer.apple.com/documentation/arkit/swiftshot_creating_a_game_for_augmented_reality`.

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

[17] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[18] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. Mobile augmented reality survey: From where we are to where we go. *IEEE Access*, 5:6917–6950, 2017.

[19] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E Culler, and Randy H Katz. MARVEL: Enabling mobile augmented reality with low energy and low latency. *ACM Sensys*, 2018.

[20] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. *ACM SenSys*, 2015.

[21] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, et al. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017.

[22] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. *ACM MobiSys*, 2010.

[23] Tuan Dao, Amit K Roy-Chowdhury, Harsha V Madhyastha, Srikanth V Krishnamurthy, and Tom La Porta. Managing redundant content in bandwidth constrained wireless networks. *IEEE/ACM Transactions on Networking (TON)*, 25(2):988–1003, 2017.

[24] R. Dubé, A. Gawel, H. Sommer, J. Nieto, R. Siegwart, and C. Cadena. An online multi-robot slam system for 3d lidars. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1004–1011, Sep. 2017.

[25] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006.

[26] Thomas Olsson et al. Expected user experience of mobile augmented reality services: A user study in the context of shopping centres. *Personal Ubiquitous Comput.*, 17(2):287–304, February 2013.

[27] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[28] Christian Forster, Zichao Zhang, Michael Gassner, Manuel Werlberger, and Davide Scaramuzza. Svo: Semidirect visual odometry for monocular and multicamera systems. *IEEE Transactions on Robotics*, 33(2):249–265, 2016.

[29] Dorian Gálvez-López and J. D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, October 2012.

[30] G. Gan and J. Cheng. Pedestrian detection based on hog-lbp feature. In *International Conference on Computational Intelligence and Security*, pages 1184–1187, Dec 2011.

[31] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[32] Google. Arcore overview. `https://developers.google.com/ar/discover/`.

[33] Google. Cloudanchor. `https://developers.google.com/ar/develop/java/cloud-anchors/quickstart-android`.

[34] Google. Snapchat lenses. `https://www.snapchat.com/`.

[35] Google. Share ar experiences with cloud anchors. `https://developers.google.com/ar/develop/java/cloud-anchors/cloud-anchors-overview-android`, May 2018.

[36] Google. Recognize and augment images. `https://developers.google.com/ar/develop/unity/augmented-images/`, 2019.

[37] Google Creative Labs. Just a Line - Draw Anywhere, with AR. `https://justaline.withgoogle.com/`.

[38] Gautam Goswami. Council Post: Augmented Reality's Applications And Future In Business. `https://www.forbes.com/sites/forbescommunicationscouncil/2020/10/15/augmented-realitys-applications-and-future-in-business/#333706602b3c`.

[39] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. *ACM MobiSys*, 2014.

[40] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *ACM Mobisys*, 2016.

[41] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[42] Sean Hollister and Rebecca Fleenor. How pokemon go affects your phone's battery life and data. `https://www.cnet.com/how-to/pokemon-go-battery-test-data-usage/`.

[43] Richard L Holloway. Registration error analysis for augmented reality. *Presence: Teleoperators & Virtual Environments*, 6(4):413–432, 1997.

[44] Andrew Howard. Multi-robot simultaneous localization and mapping using particle filters. *I. J. Robotic Res.*, 25:1243–1256, 12 2006.

[45] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[46] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *IEEE CVPR*, 2017.

[47] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. *ACM MobiSys*, 2017.

[48] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *ACM WearSys*, 2016.

[49] IKEA. Ikea place app. `https://www.ikea.com/au/en/customer-service/mobile-apps/say-hej-to-ikea-place-pub1f8af050`.

[50] Michael Irving. Horus wearable helps the blind navigate, remember faces and read books. `http://newatlas.com/horus-wearable-blind-assistant/46173/`, 2016.

[51] S. Jafri and R. Chellali. A distributed multi robot slam system for environment learning. In *IEEE Workshop on Robotic Intelligence in Informationally Structured Space*, 2013.

[52] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Overlay: Practical mobile augmented reality. *ACM MobiSys*, 2015.

[53] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Low bandwidth offload for mobile AR. *ACM CoNEXT*, 2016.

[54] Amit Jindal, Andrew Tulloch, Ben Sharma, Bram Wasti, Fei Yang, Georgia Gkioxari, Jaeyoun Kim, Jason Harrison, Jerry Zhang, Kaiming He, Orion Reblitz-Richardson, Peizhao Zhang, Peter Vajda, Piotr Dollar, Pradheep Elango, Priyam Chatterjee, Rahul Nallamothu, Ross Girshick, Sam Tsai, Su Xue, Vincent Cheung, Yanghan Wang, Yangqing Jia, and Zijian He. Enabling full body ar with mask r-cnn2go. `https://research.fb.com/enabling-full-body-ar-with-mask-r-cnn2go/`, January 2018.

[55] Li Jinyu, Yang Bangbang, Chen Danpeng, Wang Nan, Zhang Guofeng, and Bao Hujun. Survey and evaluation of monocular visual-inertial slam algorithms for augmented reality. *Virtual Reality & Intelligent Hardware*, 1(4):386–410, 2019.

[56] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 817–825, 2016.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[58] Kyoung Shin Park and R. V. Kenyon. Effects of network characteristics on human performance in a collaborative virtual environment. In *IEEE Virtual Reality*, Mar. 1999.

[59] Steven LaValle. *Virtual Reality*. Cambridge University Press.

[60] P. Li, T. Qin, B. Hu, F. Zhu, and S. Shen. Monocular visual-inertial state estimation for mobile augmented reality. In *IEEE ISMAR*, pages 11–21, 2017.

[61] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. *ACM MobiCom*, 2019.

[62] Qiang Liu and Tao Han. Dare: Dynamic adaptive mobile augmented reality with edge computing. *IEEE ICNP*, 2018.

[63] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016.

[64] Zida Liu, Guohao Lan, Jovan Stojkovic, Yunfan Zhang, Carlee Joe-Wong, and Maria Gorlatova. Collabar: Edge-assisted collaborative image recognition for mobile augmented reality. In *ACM/IEEE IPSN*, 2020.

[65] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. *IJCAI*, 1981.

[66] Tim Merel. The reality of vr/ar growth. `https://techcrunch.com/2017/01/11/the-reality-of-vrar-growth/`, January 2017.

[67] Microsoft. Shared experiences in unity. `https://docs.microsoft.com/en-us/windows/mixed-reality/shared-experiences-in-unity`, March 2018.

[68] Jannis Moeller. jannismoeller/vins-mobile-android. `https://github.com/jannismoeller/VINS-Mobile-Android`, Jul 2019.

[69] Mojang. Minecraft for Android. `https://www.minecraft.net/en-us/store/minecraft-android/`.

[70] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[71] Diala Naboulsi, Marco Fiore, Stephane Ribot, and Razvan Stanica. Large-scale mobile traffic analysis: a survey. *IEEE Communications Surveys & Tutorials*, 18(1):124–161, 2016.

[72] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. *ACM MobiSys*, 2017.

[73] Niantic. Pokemon Go. `https://www.pokemongo.com/en-us/`.

[74] Greg Nichols. `https://www.zdnet.com/article/california-firefighters-use-augmented-reality-in-battle-against-record-breaking-in`, 2018.

[75] Siddhant Prakash, Alireza Bahremand, Linda D Nguyen, and Robert LiKamWa. GLEAM: An Illumination Estimation Framework for Real-time Photorealistic Augmented Reality on Mobile Devices. *ACM MobiSys*, 2019.

[76] T. Qin, P. Li, and S. Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, Aug 2018.

[77] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *ACM MobiSys*, 2011.

[78] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam. Image change detection algorithms: A systematic survey. *Trans. Img. Proc.*, 14(3):294–307, March 2005.

[79] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering deep learning to mobile devices via offloading. *ACM Sigcomm Workshop on VR/AR Network '17*, 2017.

[80] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. *IEEE INFOCOM*, 2018.

[81] Xukan Ran, Carter Slocum, Maria Gorlatova, and Jiasi Chen. Sharear: Communication-efficient multi-user mobile augmented reality. In *ACM HotNets Workshop*, pages 109–116, 2019.

[82] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *IEEE CVPR*, 2017.

[83] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *NIPS*, 2015.

[84] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018.

[85] Mahadev Satyanarayanan, Victor Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 2009.

[86] Dieter Schmalstieg and Tobias Hollerer. *Augmented reality: principles and practice.* Addison-Wesley Professional, 2016.

[87] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. *IEEE International Conference on Robotics and Automation*, 2017.

[88] Apple App store developers. Apple app store. `https://www.apple.com/augmented-reality/`.

[89] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[90] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *IEEE CVPR*, 2014.

[91] Tiantu Xu, Guohui Wang, and Felix Xiaozhu Lin. Practical urban localization for mobile ar. In *ACM HotMobile Workshop*, 2020.

[92] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *ACM/IEEE Symposium on Edge Computing*, 2017.

[93] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. *ACM SIGCOMM*, 2015.

[94] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX NSDI*, 2017.

[95] Wenxiao Zhang, Bo Han, and Pan Hui. On the networking challenges of mobile augmented reality. *ACM SIGCOMM Workshop on VR/AR Network*, 2017.

[96] Wenxiao Zhang, Bo Han, and Pan Hui. Jaguar: Low Latency Mobile Augmented Reality with Flexible Tracking. *ACM Multimedia*, 2018.

[97] Wenxiao Zhang, Bo Han, Pan Hui, Vijay Gopalakrishnan, Eric Zavesky, and Feng Qian. Cars: Collaborative augmented reality for socialization. *ACM HotMobile*, 2018.

[98] Feng Zheng. *Spatio-temporal registration in augmented reality.* PhD thesis, The University of North Carolina at Chapel Hill, 2015.

[99] Y. Zheng, Y. Kuang, S. Sugimoto, K. Åström, and M. Okutomi. Revisiting the pnp problem: A fast, general and optimal solution. In *IEEE ICCV*, pages 2344–2351, 2013.

[100] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. In *CVPR*, volume 1, page 3, 2017.

[101] Zoran Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *International Conference Pattern Recognition*, ICPR, pages 28–31. IEEE, 2004.

[102] Zoran Zivkovic and Ferdinand van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, 27(7):773 – 780, 2006.

[103] Danping Zou and Ping Tan. Coslam: Collaborative visual slam in dynamic environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(2):354–366, 2012.