UNIVERSITY OF CALIFORNIA
RIVERSIDE

Energy and Network Aware Mobile Augmented Reality

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

Doctor of Philosophy

in

Computer Science

by

Kittipat Apicharttrisorn

September 2021

Dissertation Committee:

Dr. Srikanth V. Krishnamurthy, Chairperson
Dr. Amit K. Roy-Chowdhury
Dr. Evangelos Papalexakis
Dr. Jiasi Chen

The Dissertation of Kittipat Apicharttrisorn is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank the University of California, Riverside, and the Department of Computer Science and Engineering for the facilities and resources that are provided to me throughout my PhD study. I also feel privileged to take the courses taught by insightful professors who help expand my intellectual horizons with the knowledge that is both important and relevant. In addition, I am incredibly grateful to my advisors, Professor Srikanth Krishnamurthy and Professor Jiasi Chen for tirelessly training and preaching me to be a better scientist everyday. Their mastery and dedication to scientific research are a blueprint for my research career now and in the future.

I am also honored to be part of a vibrant, cheerful and talented body of graduate students – Shailendra Singh, Tuan Dao, Azeem Aqil, Pravallika Devineni, Ahmed Atya, Yue Cao, Daimeng Wang, Ahmad Darki, Saheli Ghosh, Shasha Li, Shitong Zhu, Xukan Ran, Hisham A. Alhulayyil, Yi-Zhen Tsai, and others not mentioned here due to space contraint. I not only know them personally but also have chances to see them succeed in the endeavors they turn to, and this motivates me to work harder and harder everyday.

Finally, this dissertation would not be possible without love and support from my parents – Chaiwitoon and Arpa Apicharttrisorn, and from all my family with the last names – Apicharttrisorn, Fadklang, and Taechareon. This dissertation is also dedicated in loving memory of my sister Narumol Taechareon. Importantly, I would like to thank my wife, Dusadee Apicharttrisorn, for her endless support during the work toward this dissertation.

To my parents for all their unconditional love.

ABSTRACT OF THE DISSERTATION

Energy and Network Aware Mobile Augmented Reality

by

Kittipat Apicharttrisorn

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2021
Dr. Srikanth V. Krishnamurthy, Chairperson

This dissertation has two main objectives – solving power and latency issues in mobile augmented reality. For power, we showcase the power drain due to the two heaviest components – simultaneous localization and mapping (SLAM) and deep convolutional neural networks (DNNs) and design solutions to reduce the power consumption on mobile devices. Our single-user solution is to use DNNs as needed, to detect new objects or recapture objects that significantly change in appearance, and otherwise depend on low-power object tracking. For multi-user solutions, we use peer-to-peer communications to exchange key information among devices, and finally assign roles to each of them – primary or secondary. A primary device continuously tracks target objects and shares their information to slaves. Secondary devices do not need SLAM or DNN but leverage the shared information from the master and other lightweight methods to keep track of the objects with high precision, and thus significantly reduce power consumption. In addition, we can rotate the master functionality across participants in order to distribute energy expenditures among them and increase the longevity of the AR experience. For latency, we perform a first-of-its-kind measurement study on both public LTE and industry LTE testbed for two popular multi-user AR applications, yielding several insights such as: (1) The radio access network (RAN) accounts for a significant fraction of the end-to-end latency (31.2%, or 3.9 s median); (2) AR network traffic is characterized by large intermittent spikes on a single uplink TCP connection, resulting in frequent TCP slow starts that can increase user-perceived latency; (3) Applying a common traffic management mechanism of cellular operators, QoS Class Identifiers (QCI), can help by reducing AR latency by 33% but impacts non-AR users. Based on these insights, we propose AR solutions to intelligently adapt IP packet sizes and periodically provide information on uplink data availability, respectively. Our solutions help ramp up network performance, improving the end-to-end AR latency and goodput by 40-70%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile augmented reality (AR) is gaining popularity among users with applications in gaming, education, medical or emergency responses because it allows them to see virtual holograms or virtual objects attached to their physical worlds. There are two important problems in mobile augmented reality – latency and power. For the former, long latency among multiple users in AR causes inconsistencies among them; for example, a user may already remove a virtual object but another user is still seeing and interacting with it. For the latter, AR applications process the visual information using algorithms such as Deep Neural Networks (DNN) in order to detect physical objects in the Field of Views (FoV) or Simultaneous Localization and Mapping (SLAM) in order to estimate the surroundings and the device's position relative to those surroundings. AR devices will drian the batteries if running these algorithms continuously without control.

In this dissertation, we address the problems and propose the solutions as follows.

- Augmented reality (AR) apps where multiple users interact within the same physical space are gaining in popularity (*e.g.,* shared AR mode in Pokemon Go, virtual graffiti in Google's Just a Line). However, multi-user AR apps running over the cellular network can experience very high end-to-end latencies (measured at 12.5 s median on a public LTE network). To characterize and understand the root causes of this problem, we perform a first-of-its-kind measurement study on both public LTE and industry LTE testbed for two popular multi-user AR applications, yielding several insights: (1) The radio access network (RAN) accounts for a significant fraction of the end-to-end latency (31.2%, or 3.9 s median), resulting in AR users experiencing high, variable delays when interacting with a common set of virtual objects in off-the-shelf AR apps; (2) AR network traffic is characterized by large intermittent spikes on a single uplink TCP connection, resulting in frequent TCP slow starts that can increase user-perceived latency; (3) Applying a common traffic management mechanism of cellular operators, QoS Class Identifiers (QCI), can help by reducing AR latency by 33% but impacts non-AR users. Based on these insights, we propose network-aware and network-agnostic AR design optimization solutions to intelligently adapt IP packet sizes and periodically provide information on uplink data availability, respectively. Our solutions help ramp up network performance, improving the end-to-end AR latency and goodput by ∼40-70%. Details can be found in Chapter 2.

- Accurate tracking of objects in the real world is highly desirable in Augmented Reality (AR) to aid proper placement of virtual objects in a user's view. Deep neural networks (DNNs) yield high precision in detecting and tracking objects, but they are energy-heavy and can thus be prohibitive for deployment on mobile devices. Towards reducing energy drain while maintaining good object tracking precision, we develop a novel software framework called MARLIN. MARLIN only uses a DNN as needed, to detect new objects or recapture objects that significantly change in appearance. It employs lightweight methods in between DNN executions to track the detected objects with high fidelity. We experiment with several baseline DNN models optimized for mobile devices, and via both offline and live object tracking experiments on two different Android phones (one utilizing a mobile GPU), we show that MARLIN compares favorably in terms of accuracy while saving energy significantly. Specifically, we show that MARLIN reduces the energy consumption by up to 73.3% (compared to an approach that executes the best baseline DNN continuously), and improves accuracy by up to 19× (compared to an approach that infrequently executes the same best baseline DNN). Moreover, while in 75% or more cases, MARLIN incurs at most a 7.36% reduction in location accuracy (using the common IOU metric), in more than 46% of the cases, MARLIN even improves the IOU compared to the continuous, best DNN approach. Details can be found in Chapter 3.

- Collaborative AR applications are gaining popularity (*e.g.,* Pokemon Go for entertainment, AURORA for military battlefields), but have heavy computing requirements. Prior AR frameworks typically rely on dedicated infrastructure such as edge computing to offload AR's compute-heavy tasks. However, such infrastructure may not always be available (*e.g.,* in battlefields), and continuously running AR computations on user devices can rapidly drain battery and impact application longevity. In this work, we enable infrastructure-free mobile AR with a low energy footprint, by using *collaborative time slicing* to distribute compute-heavy AR tasks across user devices. Realizing this idea is challenging because distributed execution can result in inconsistent synchronization of the AR holograms. Our framework COLLAR tackles this with novel lightweight techniques for tight synchronization between users and low latency recovery upon disruptions. We prototype COLLAR on Android and show that it can both reduce power by up to 36% and improve the hologram positioning accuracy (with respect to the IOU metric) by up to 80%, relative to state-of-the-art AR systems. Details can be found in Chapter 4.

Chapter 2

Characterization of Multi-User Mobile Augmented Reality over Cellular Networks

2.1 Introduction

Augmented reality (AR), with its premise of virtual objects integrated with our physical environment, promises new immersive experiences, and the market is forecast to reach 100 billion dollars by 2021 [76, 80]. AR applications such as navigation, entertainment (*e.g.,* Pokemon Go), and field service involve multiple users, co-located in the same shared outdoor environments, relying on low latency communication over the cellular network to obtain a consistent view of virtual objects. In this chapter, we consider scenarios where multiple users are co-located in the same real physical space, and wish to view a common set

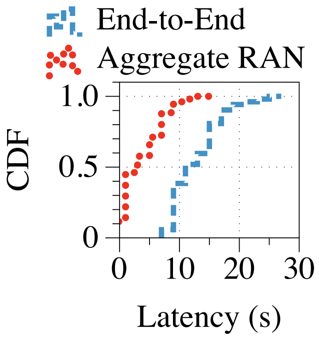


Figure 2.1: Multi-User AR latency

of virtual objects. Our measurements of such off-the-shelf AR apps over cellular networks show that the user-perceived end-to-end AR latencies are extremely high. For example, Fig. 2.1 shows the CDF of the AR latencies of the Google CloudAnchor AR app [43], running on a 4G LTE production network of a Tier-I US cellular carrier at different locations and times of day (details in §2.4). The results show a median 3.9 s and 12.5 s of aggregate radio access network (RAN) latency and end-to-end AR latency (as explained below), respectively. As latency is a key contributor to AR quality-of-experience (QoE) [32], a deeper understanding of the root causes of these high latencies is needed in order to improve AR's end-to-end performance over cellular networks, which is the key focus of our paper.

In this context, end-to-end AR latency is the total time from when one User Equipment (UE) places a virtual object in the real world until when another UE can view the object on her screen. Aggregate RAN latency is defined as the subset of this time for over-the-air transmissions.

Why is AR different?: While the cellular network is relatively well equipped to handle traditional applications such as web and video, AR presents new challenges because of its unique application and communication characteristics, as discovered in this chapter. In brief, AR differs from other multimedia applications such as video or 360° Virtual Reality (VR) streaming, short-form video uploads (*e.g.,* Snapchat or Instagram Stories) and video conferencing in the following key ways:

- *Lack of playback buffers and latency-sensitivity*: In video and 360° VR streaming, the length of the playback buffer, which caches the yet-to-be-played video chunks for the player, impacts streaming patterns and traffic burst periods. We observe that off-the-shelf AR apps do not continuously upload data, and they consume AR data holistically, unlike in video/VR, where the player consumes data frame-by-frame. Hence, AR data objects need to be delivered *quickly* to the AR device (user equipment, or UE), in order to avoid latency-based QoE issues. While video conferencing similarly lacks playback buffers and has tight latency deadlines, delayed frames can be skipped or played back quickly later, whereas delayed AR transmissions can lead to inconsistent user manipulations of the virtual objects (*e.g.,* user A touches a non-existent virtual object that has already been moved by user B).

- *Lack of application adaptation mechanisms*: Video and 360° VR streaming make use of adaptive bit rate (ABR) mechanisms, such as MPEG-DASH, which adapt the streaming resolutions of the video chunks to avoid video QoE issues. However, off-the-shelf AR currently does not use ABR, and cannot be modified to do so in closed source commercial AR systems. Additionally, even if such systems were transparent, it is unclear how such application-layer adaptations should be done in AR. (discussed in §2.5.3)

- *Uplink-heavy TCP traffic*: In multi-user gaming, the traffic is comprised of small uplink UDP [105] data (mainly from user movements/actions) with stringent latency requirements. For short-form video uploads (such as Instagram Stories or Snapchat), live video upload or conferencing, even though the traffic is uplink-heavy, latency tolerance is higher than AR [32]. Live or buffered video streaming apps such as YouTube use downlink QUIC [67] instead of TCP, and are more latency-tolerant than AR. In contrast, we observe that AR network traffic is different from all these, since it is uplink-heavy, TCP-based and latency-sensitive. Hence, TCP performance for the AR session is critical to its end-to-end latency and throughput, and we investigate its interactions with the RAN and AR in this work.

Contributions: Motivated by these unique characteristics of multi-user AR, we perform the first detailed experimental study across the application, IP, and RAN layers to characterize how the cellular network impacts AR applications. We provide crucial insights on cross-layer inter-dependencies involved in multi-user AR streaming. Existing work on AR either focuses on real object detection with cloud/edge support [58, 92, 76] over WiFi, or efficient device localization [70] while neglecting the communication aspects. Works involving multiple AR devices [93, 125, 87] focus on application layer performance, without quantifying the interactions between AR application, the cellular network, and cloud processing.

Our main contribution in this chapter is a measurement-driven characterization of multi-user AR on both (i) public 4G LTE cellular carriers of a Tier-I US mobile network operator to capture realistic network, RF, and traffic conditions, and (ii) an experimental

LTE industry testbed with fully-implemented RAN protocol stack and virtual EPC that allows us to vary network settings (such as cell load, radio bearer QoS class, etc.) for controlled testing, in order to quantify their impact on AR performance. We study widely-used AR apps in the market utilizing an off-the-shelf AR platform, Google ARCore [44]. This platform is broadly representative since it provides multi-user AR capabilities in Android devices and analogous APIs are provided by Apple ARKit [15], Microsoft Hololens [82], and Magic Leap [69] (see §2.3 for methodology details).

Our measurement study leads to several insights:

- To quantify differences between multi-user AR and other multimedia applications, we compare the user-perceived latencies across these apps in §2.4.1. Then, we provide a component-wise breakdown of the end-to-end (E2E) AR latency and show that the RAN accounts for ∼31.2% of the overall latency on average over public LTE (∼3.9s). This causes the AR devices to experience high delays when interacting with the virtual objects (§2.4.2). We also characterize the performance of RAN optimization techniques such as QoS Class Index (QCI) adaptation to serve AR traffic. While QCI adaptation improves the E2E latency for AR users by ∼33%, it reduces the throughput of non-AR users by ∼31.6% (§2.4.4).

- We show that AR traffic is bursty with large time gaps between successive bursts of uplink data (*e.g.,* 20s on average for CloudAnchor with burst sizes of ∼2.5 MB) whenever a virtual object is placed. This causes the TCP congestion window (`cwnd`) to enter slow-start before the beginning of each burst. We show that RAN segmentation latency at the RLC layer significantly impacts TCP performance, during the slow-start phase (§2.4.5).

- We propose a network-aware AR app design optimization technique that intelligently adapts IP packet sizes for the AR app, based on the underlying RAN conditions. Our methodology in selecting packet sizes addresses the trade-off between minimizing segmentation of packets at the RAN (caused by larger IP packets), which adversely impacts network latency, and minimizing network overhead (caused by smaller IP packets), which adversely impacts application goodput. Our technique improves the aggregate RAN latency, end-to-end AR latency, network throughput and application goodput by ∼40-70% (§2.5.1).

- We propose a network-agnostic AR app design optimization technique that periodically updates the LTE base station (eNB) of the uplink RAN buffer status of the hosting AR device, even during gaps between AR bursts. We achieve this by generating negligible, periodic amounts of dummy data, which enables constant UE buffer status updates to the eNB. This results in improved RAN resource allocation for the AR device and minimizes uplink signaling latencies. Our technique improves the aggregate RAN latency by ∼50%, at a marginal cost of additional bandwidth (§2.5.2).

Since existing AR apps are closed-source and their data transmissions being opaque, our work focuses on network-layer characterizations and solutions. However, we provide a brief discussion on how one can adapt the AR application content to reduce latency, potentially with other user-perceived performance costs that may be acceptable (§2.5.3).

We release our RAN latency analysis tool, which runs on client devices (UEs), as open source [11]. It can be used by researchers with data captured on public LTE networks, without any modifications needed to the eNB.

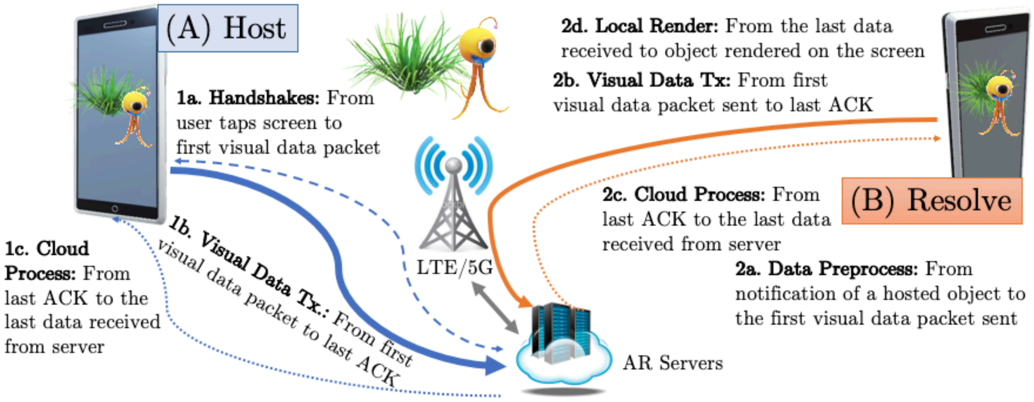

(A) Host
1a. Handshakes: From user taps screen to first visual data packet
1b. Visual Data Tx.: From first visual data packet to last ACK
1c. Cloud Process: From last ACK to the last data received from server
LTE/5G
AR Servers
2d. Local Render: From the last data received to object rendered on the screen
2b. Visual Data Tx: From first visual data packet sent to last ACK
2c. Cloud Process: From last ACK to the last data received from server
(B) Resolve
2a. Data Preprocess: From notification of a hosted object to the first visual data packet sent

Figure 2.2: Cloud-based multi-user AR. Use of the cloud is mandated for multi-user AR apps in Android.

2.2 AR Background and Related Work

Background on AR: When current AR devices (*e.g.,* those running the Apple ARKit, Google ARCore, or Microsoft Hololens AR platforms) wish to place virtual objects in the real world, they first perform device localization in order to create a consistent 3D coordinate system of the real world. The real-world coordinate system (called the *world frame*) provides a common reference for the devices to place the virtual objects, and is constructed using Simultaneous Localization and Mapping (SLAM) techniques [101]. Once the devices have a common world frame and know the poses (location and orientation) of the virtual objects, the virtual objects can be drawn on each device's display when it is within the user's field-of-view. Below we describe the steps involved in synchronizing the world frame between device A (which places a virtual object), and device B (which receives and renders that virtual object), as illustrated in Fig. 2.2.

1. **Hosting (device A):** (a) ***Handshakes***: Device A initiates connections with the cloud: a Firebase database, and two Google Cloud instances for visual positioning.

(b) ***Visual Data Tx***: Device A sends real-world visual data and information about the virtual objects (position, orientation, 3D sprite/texture maps) to the cloud. (c) ***Cloud Process***: The cloud processes A's visual data using SLAM to compute the world frame.

2. **Resolving (device B):** (a) ***Data Preprocess***: Device B scans and retrieves camera frames and pre-processes the data. (b) ***Visual Data Tx***: Device B sends its visual data to the cloud. (c) ***Cloud Process***: The cloud matches B's visual data against the world frame computed in step 1c, and computes B's location and orientation in the world frame. (d) ***Local Render***: B uses the information from the cloud to render the virtual object at the correct position and orientation on the display.

Related Work: To the best of our knowledge, we are the first to perform an in-depth measurement study of AR applications operating on cellular networks.

Mobile AR: Many works study object detection for single-user AR [76, 92, 77, 124, 30, 12], with cloud/edge processing to reduce latency and/or energy. A few papers [125, 87, 93] discuss multi-user AR with focus on the application-layer sharing. In this work, we focus on SLAM-based AR, prevalent in off-the-shelf AR systems, and the communication aspects of multi-user AR when operating on cellular networks.

Multi-user SLAM: Some work has been done on multi-user SLAM in the robotics context [130, 102]. These works mainly focus on the SLAM algorithms themselves, and not their communication aspects.

QoS for cellular networks: Work on service-level QoS for the cellular network allocates physical resource blocks (PRBs) for users through smart eNB schedulers, QCI

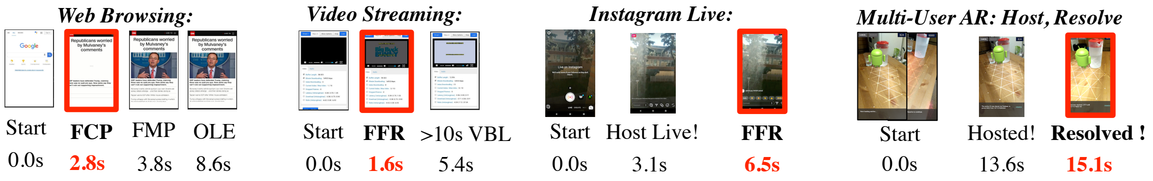


Figure 2.3: Application screenshots, red-boxed screens are user-perceived events (FCP: First Contentful Paint, FMP: First Meaningful Paint, OLE: onLoad Event, FFR: First Frame Rendered, VBL: Video Buffer Length)

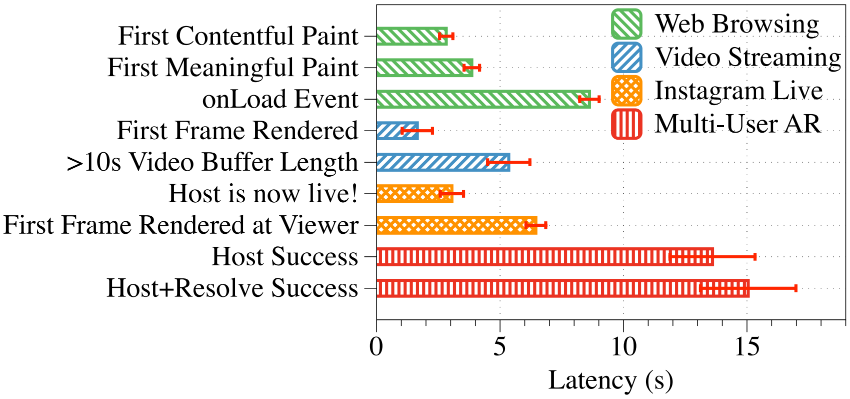


Figure 2.4: User-perceived latency

selection, or combinations thereof [4, 60, 6]. However, naively applying these techniques may not work well for AR's bursty traffic patterns (§2.4.4).

2.3 Methodology, Testbeds, and Tools

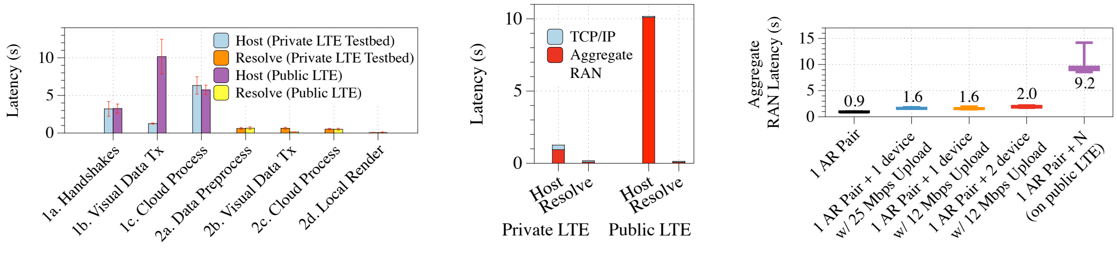
Multi-User AR apps: We investigate multi-user AR with the *Cloud Anchor* [43] and *Just a Line* [41] demo apps provided by Google. *Cloud Anchor* allows one user to place a virtual object in the scene and a second user to view it. *Just a Line* allows two users to draw virtual graffiti in a shared physical space. These apps all rely on Google's CloudAnchor

API [39], which is a key API to provide multi-user capabilities for Android AR apps. Thus, our observations across different apps are corroborated as they all rely on this fundamental API. The experiments in this chapter are done using Cloud Anchor, except where it is specified otherwise.

Experimental setup: (a) *Industry LTE testbed:* We use an in-house outdoor 10MHz LTE testbed (operating on Band 30, 2.3 GHz) with a virtual EPC core and an LTE eNB, having 2 LTE cells with 2×2 MIMO capability. Each cell yields peak uplink and downlink rates of 25 Mbps and 50 Mbps, respectively. The AR UE pair (hosting UE, rendering UE) and the load phones are connected to the eNB. We use a pair of OnePlus 5T phones as AR test devices and Samsung Galaxy S7 phones as load UEs to provide background traffic. The phones can support 2x2 MIMO. We varied the RF conditions of the AR UEs resulting in uplink SINR values ranging from $5 - 17$ dB and RSRP values ranging from -85 dBm to -105 dBm. The eNBs are running on HP380 servers with WindRiver Linux and transmission power of 26 dBm using 2T2R antennas. The eNBs are connected to the vEPC core, and further to the public IP network. The testbed also allows controlling the user and traffic load on the cells, and modifying parameters like QoS Class Identifier (QCI) for service differentiation of traffic classes (see Sec. 2.4.4).

(b) *Public LTE network:* We perform experiments over a public 20 MHz LTE network with 2× 2 MIMO and Carrier Aggregation capability on a Tier-I US carrier. We use a pair of Google Pixel 2 phones as AR test devices[1]. We perform experiments on public LTE measurements with RSRP and SINR values of the AR UEs ranging from -85 to

[1] Different UE device models do not make a significant difference in the experiments because AR cloud servers perform the heavy computations and AR UEs only send data to the cloud and render virtual objects.



(a) Latency breakdown

(b) RAN latency

(c) Impact of other users

Figure 2.5: Breakdown of end-to-end latency. Visual data transmission and RAN latencies are significant.

-110 dBm and 5 − 17 dB, respectively. We perform measurements in different locations: a campus cafeteria, a public mall, downtown and residential areas, and commercial business, with 5-12 trials per location during daytime or nighttime.

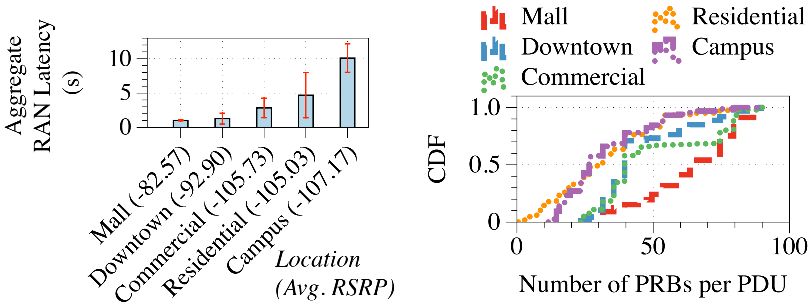
Measurement Tools: 1) *Application layer*: We instrument the AR apps to synchronize and log Unix timestamps of application events. For web and video applications, we profile the latencies on Android devices using Chrome's remote debugging developer tools [18]. 2) *TCP/IP layer*: We use `tcpdump` to capture IP packets with timestamps. 3) *RAN layer*: We measure the RAN latency by running MobileInsight [71] (MI) to capture LTE PDUs on the test UEs. We develop a custom analyzer [11] to parse the MI logs, extract PDU-level information and compute RAN latency.

2.4 Measurements of AR Over Cellular Networks

2.4.1 Application-layer performance

AR streaming vs other applications: Latency is a key metric for multi-user AR. If the users experience disparate latencies, consistency issues may result, such as one user placing a virtual object and another user not being able to view it quickly, or one user attempting to manipulate a virtual object that has already been moved by another user.

From Fig. 2.1, we have shown that the end-to-end latency has a very high median value of 12.5 s, resulting in poor QoE for the AR user [32]. Here, we discuss why other applications, such as web, on-demand video and live streaming uploads over cellular networks, do not suffer from similar QoE problems. For this, we conduct a set of experiments where a user surfs `www.cnn.com`, streams an MPEG-DASH video, hosts a live video stream on Instagram, and plays multi-user AR on public LTE (all experiments were conducted in sequence within a one-hour duration). Fig. 2.3 shows screenshots from the perspective of the user, and Fig. 2.4 shows the latency of each application-level event. For web browsing, although the complete content is only loaded (`onLoad` event) 8.6 s after the user starts browsing , the first contentful paint and first meaningful paint happen at 2.8 s and 3.8 s respectively. Similarly, for on-demand video, the user sees the first frame rendered on the screen after 1.6 s, due to video rate adaptation by MPEG-DASH. On-demand video also has playback buffers for pre-fetching video chunks and so is not latency-critical, as AR is. Similar video rate adaptation mechanisms apply to Instagram Live. The stream goes live within 3.1s (as notified by the server), and a viewer can view the first frame 3.4s later (6.5s after the host started the stream).



(a) Aggregate RAN latencies at five locations

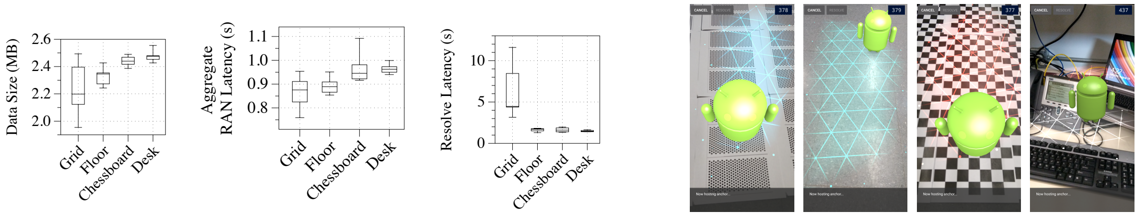
(b) Uplink physical resources

Figure 2.6: CloudAnchor measurements at five different locations.

In summary, even though it takes 5-8s to download the full web or video content, user experience is not impacted because web and video have application-layer adaptation mechanisms (*e.g.,* paint the visible part of the webpage as soon as possible, adapt the video quality to have a low time-to-first render, or cache frames in the video buffer). In contrast, in multi-user AR, which lacks application-layer adaptation mechanisms, the resolving user can only see the content after the hosting user finishes its entire data transmission, which takes ∼15s.

2.4.2 Breakdown of end-to-end latency

Latency breakdown: Having shown the detrimental impact of latency on AR QoE, we seek to understand the key contributors to the high end-to-end latencies observed in Sec. 2.4.1. We plot the constituent components of the latency in Fig. 2.5a, descriptions of which are provided in Sec. 2.3. On our industry LTE testbed, we see that the key latency contributors are on the hosting side (device A): the handshakes, the visual data transmission,



(a) Data size, host RAN latency, resolve latency

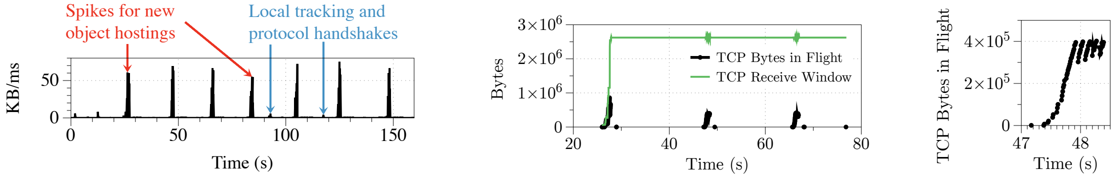
(b) Real-world environments: grid, floor, chessboard, desk

Figure 2.7: Hosting augmented objects in a grid environment results in the smallest data sizes and RAN latency, but requires multiple attempts to resolve the virtual object.

and the cloud processing all together consume 10s on average (86.6% of the total end-to-end latency), while the resolving side (device B) includes data preprocessing, visual data transmission, cloud processing, and local rendering, and are relatively quick (12.9% of end-to-end latency). On public LTE, the hosting steps takes 93.3% of the end-to-end latency because of its long visual data transmission latency. The visual data transmission is the largest contributor to latency on public LTE (10.1s on average, or 48.7% of the end-to-end latency), but consumes less time on the LTE testbed (17.2% of the end-to-end latency) due to lack of contention with other users (we observe similar latency on low congestion public LTE at a mall in daytime/weekday shown in Fig. 2.6a). The high transmission latencies exceed previously observed communication delays on AR research prototypes [58]; we hypothesize that this is due to information from multiple frames being uploaded, as well as additional image features such as point cloud data (full details are unknown because the off-the-shelf AR systems that we test on are closed source).

Wireless latency matters: Since we observe above that the uplink visual data transmission latency by the hosting device is significant, we further decompose the visual data transmission time into TCP/IP and radio access network (RAN) components. This provides an understanding of how much time is spent on the wireless link, and how much time is spent in the wired backbone. The aggregate TCP/IP latency across IP packets is measured as the elapsed time from the first visual data packet transmission at the TCP layer to the reception of the last packet's ACK (step 1b in Sec. 2.3). The aggregate RAN latency across IP packets is the total time from when the first visual data packet is received by the RAN layer (i.e. LTE's PDCP), until when the last packet is sent to the MAC layer for transmission. It is subsumed within the aggregate TCP/IP latency. The results are shown in Fig. 2.5b. We observe that the RAN contributes the majority of the visual data transmission time (71.7% of the visual data transmission time on LTE testbed, and 98.5% on public LTE), suggesting that the wireless link is a key contributor to end-to-end latency, especially on public LTE where the data transmission on the RAN takes an average of 10.1 s. Hence further optimizations of the wireless link are needed, as discussed in §2.4.4 and §2.5.

In addition to the above results on campus using public LTE, we repeat the measurements at four other locations with different wireless signal strengths (RSRP), as shown in Fig. 2.6a. RAN latencies at these locations range from 1-10s and seem to be correlated with the RSRP. The devices with poor signal strength tend to have fewer uplink resources allocated, as shown in Fig. 2.6b.



(a) Throughput over time.

(b) Bytes in Flight of first three bursts in 2.8a.

(c) Second burst zoom in 2.8b.

Figure 2.8: AR apps exhibit large, unpredictable data spikes on the same TCP connection, which results in TCP slow start re-triggering each spike.

Hence, our wireless link optimizations of Sec. 2.4.4, particularly the QCI-based adaptations that impact uplink resources, can potentially provide the most gains for devices with poor signal quality.

Finally, we examine the impact of background users on AR performance, and the impact of AR applications on the background users. In our industry LTE testbed, we set up background devices uploading `iPerf3` UDP traffic with finite send buffer (12 or 25 Mbps, representing 50% or 100% of the maximum uplink RAN capacity, respectively), and plot the results in Fig. 2.5c. We see one background user cause a 65.5% increase in RAN latency for the AR user and two background users cause a 111.3% increase. On the public LTE where the number of background users and their traffic are uncontrolled and unknown, the RAN latency for the AR user increases $\sim 10\times$ possibly due to high cellular network congestion.

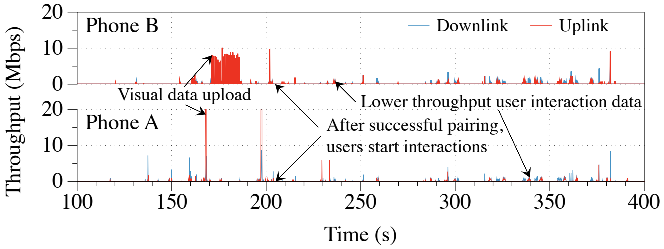


Figure 2.9: Another AR app, Just A Line, also exhibit smaller data spikes after the initial large ones, corresponding to users drawing virtual graffiti.

Resolve latency: While in the majority of cases, the visual data transmission by the hosting device (step 1b, §2.2) contributes greatly to the end-to-end latency, in a few cases, we actually observed that the virtual object resolving process can cause high delay. This is despite the small amounts of data being uploaded by the resolving device (step 2b, §2.2). We observe that this happens when the user tries to place virtual objects in real-world environments lacking visual features (*e.g.,* high-contrast edges, colors, etc.). We experimented with several real-world environments ranging from simple to complex, as shown in Fig. 2.7b. The RSSI remains relatively constant at -66 dBm. We measured the data size, uplink RAN latency, and resolve latency and plot the results in Fig. 2.7a. In the simple grid and floor environments which lack visual features, we observed relatively less data uploaded by host device A (2.2-2.34 MB on average) and thus lower uplink RAN latency (0.88-0.89 s on average). However, these simpler environments also caused high resolve latency, as shown in Fig. 2.7a. This is due to multiple rounds of communication between the device B and the cloud, unlike the typical scenario of 1-2 rounds of communication we had observed in the non-grid environments.

We hypothesize that the lack of visual features in the grid environment causes difficulties in the world frame construction (step 1c, §2.2) , resulting in these multiple rounds of communication (step 2b) as device B uploads additional visual data to aid in cloud processing (step 2c).

2.4.3 AR traffic characteristics

Bursty uplink AR traffic: To understand why and how often the visual data transmissions occur, we examine their relationship with AR user interactions (*e.g.,* placing virtual objects, drawing virtual graffiti). Fig. 2.8a shows a sample throughput trace of device A running the CloudAnchor app (other traces are similar; we show one example for brevity). The large spikes correspond to large data transmission bursts when the user touches device A's screen to place a virtual object. We observe that most of the data transmissions happen on the uplink from host device A (2.5 MB on average), while the amount of data generated on the downlink or by device B is negligible (< 100 KB). These larger data sizes contribute to the high end-to-end latencies discussed above.

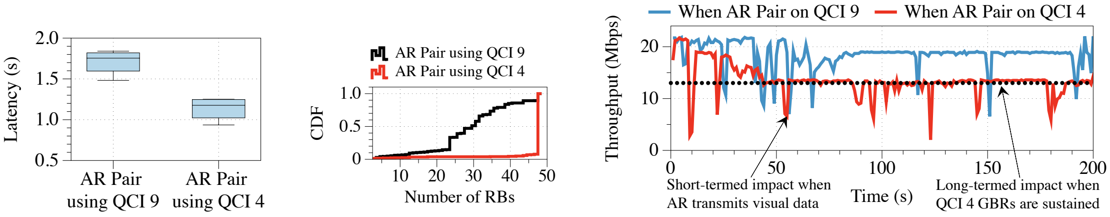
We also observe a second, smaller type of AR user interaction data, as exemplified by the throughput trace in Fig. 2.9 from the Just a Line app. Fig. 2.9 shows both the large visual data spikes near $t = 170, 200$ s, similar to those observed in the CloudAnchor app in Fig. 2.8a, and smaller bursts near $t = 230, 295, 340$ s, etc. These smaller bursts correspond to the user touching the screen drawing virtual graffiti (447 bytes of IP packet length on average), and are smaller than the initial visual data spikes (1430 bytes on average).

In summary, AR traffic has bursts of both large and small data, corresponding to different types of user interactions/scenarios. Based on our understanding of ARCore [43], we posit that the larger spikes correspond to visual data about the scene, which is necessary whenever the app is initialized or the user moves to a new location and wishes to place virtual objects, while the smaller spikes correspond to user interactions with the virtual objects after the initial visual data has been uploaded. Time delay between data spikes depends on the frequency of user interactions, which can be unpredictable, depending on the application content.

Interaction with TCP: One implication we observe from the bursty nature of AR traffic is its interaction with TCP congestion control. All the data spikes happen in the same TCP stream, and so are affected by the same receive window. In Fig. 2.8c, we plot the number of TCP bytes in flight corresponding to the second spike in Fig. 2.8b, which corresponds to the first three spikes in Fig. 2.8a. We observe that each time a data spike happens (when the AR application has visual data to send), the number of TCP bytes in flight has to grow in a slow start phase. This is because the TCP congestion window decreases when the connection is idle, in between the AR user's interactions. On the other hand, applications such as video live streaming continuously have application-layer video data ready to upload (we observed this in our experiments with *Instagram Live*), and can continuously grow the congestion window without repeatedly dropping to slow start.

2.4.4 Can dedicated QoS classes help AR?

QoS Class Identifiers (QCI) are widely used by network providers [4] to offer differentiated QoS for services, where a service is assigned to a bearer with a specific QCI value for data transmission. In our industry LTE testbed, allowing control over the QCI



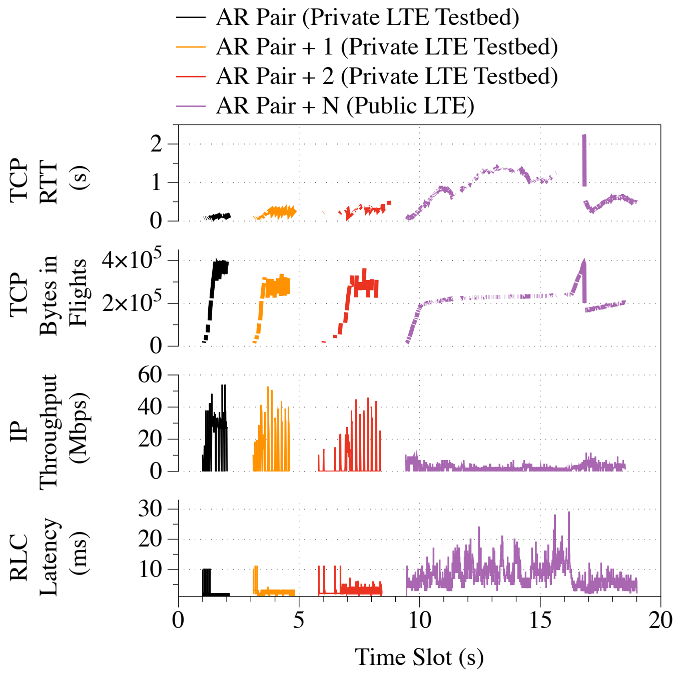
(a) IP Latency (b) PRBs (c) Impact to non-AR users

Figure 2.10: Impact of QoS dedicated bearer: Assigning QCI-4 with a guaranteed bit rate to the AR user reduces its latency, but decreases the throughput of other users, even if the AR user is not transmitting in between user interactions.

classes, we set up one AR pair and one background user uploading 25 Mbps `iperf3` traffic. The AR users are configured to use QCI-4 with a guaranteed bit rate (GBR), while the other user remains on the default, best-effort QCI-9. We configure QCI-4 to have a very high GBR of 25 Mbps (the total available bandwidth), in order to ensure that the AR users receive prioritization without any limitations.

Fig. 2.10a shows the latency of the AR user with and without QCI-4. QCI-4 helps reduce the TCP/IP latency by 33%, which represents an upper bound improvement even if more users were present. The improvement is due to resource prioritization for the AR user, because the higher priority of QCI-4 allows the AR flow to be scheduled on the majority of available PRBs in the cell, as shown in Fig. 2.10b, and across consecutive TTIs contiguously.

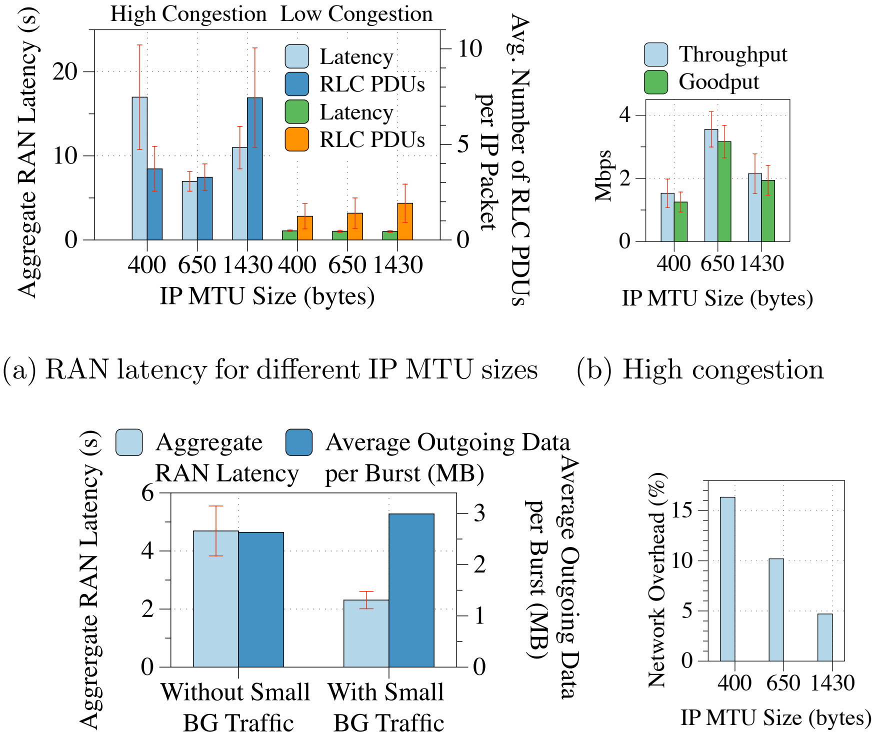
While a dedicated bearer such as QCI-4 can help reduce AR latency, Fig. 2.10c shows the performance achieved by the non-AR iPerf user. When the AR flow is assigned to QCI-9, the iPerf user can obtain most of the available bandwidth (20 Mbps), with occasional dips due to the AR user's data bursts. In contrast, when the AR flow is assigned to QCI-4, the iPerf user has its throughput reduced to an average of 13 Mbps, even when the AR user is not transmitting, because the eNB permanently reserves wireless resources for the



(a) Per-packet RLC latency

GBR user [4]. While a live-streaming video service assigned to a GBR bearer stream would continuously and predictably utilize the assigned network resources, the AR flow wastes network resources due to its bursty and unpredictable nature.

These results suggest several challenges in designing an "AR-specific" QCI class. The RAN should to be able to predict when an AR data spike is about to begin, quickly assign this flow to a dedicated bearer, estimate when the spike is about to end, and finally remove the dedicated bearer. This can prevent negative impacts to other users in the network. For example, in light of the large and small data spikes observed in Sec. 2.4.3, we may not need to keep a dedicated bearer after the large data spikes have occurred.



(a) RAN latency for different IP MTU sizes

(b) High congestion

(c) Small background traffic

(d) Network Overhead

Figure 2.12: Per-packet RLC latency is high at the beginning of the data spike, increasing RTT especially during TCP slow start. A smaller IP MTU reducing RLC segmentation and small amount of background traffic are possible solutions to help reduce AR latency.

2.4.5 Below the IP Layer: RAN Analysis

In this section, we take a detailed look at AR's behavior below the IP layer, in order to understand LTE's impact on AR performance. The IP layer passes its packets to LTE's PDCP layer, and from there to the RLC, MAC, and finally PHY layer as PDUs for transmission. Channel conditions and traffic loads generated by all the users determine the size of the RLC PDU in the current scheduling period for a given device. Based on the PDU size, the RLC layer then performs an important operation: it concatenates or segments the IP packets to fit into the RLC PDU(s), which is a key contributor to RAN latency.

Fig. 2.11a illustrates the relationship between per-packet RLC latency and IP throughput for the similar test cases as §2.4.2. *AR pair (+ #load phones)* where load phones generate 12 Mbps finite-buffer traffic are performed on our private LTE testbed while *AR pair + N* trails are done in public LTE with unknown traffic and number of load phones. Across test cases, we observe that the TCP RTT (first row) increases with RLC latency (bottom row), especially in the public LTE test case. The longer RTT can cause the TCP congestion window to ramp up slowly. This is shown by the relatively smaller rate of growth in the number of TCP bytes in flight over time (second row), especially during the slow-start phase when the RLC latency is higher, subsequently resulting in reduced IP throughputs (shorter and sparser lines in the third row). The impact of RLC latency on TCP slow start is crucial in AR because AR can be prone to frequent slow start phases due to the time gap between user interactions, as discussed in Sec. 2.4.3. The relationship between RLC latency, TCP RTT, and IP throughput suggests that RLC latency is an important factor to increase the throughput and improve the latency of AR applications.

2.5 AR Design Optimizations

In this section, based on the insights gleaned from the traffic characterization in §2.4, we provide AR design optimizations.

2.5.1 Network-Aware Optimization: Packet Size Adaptation

When the AR app uses larger IP packet sizes for transmission, it could experience heavier segmentation at the RLC layer, especially when the RLC PDU sizes are significantly smaller than the packet sizes. This happens in scenarios when the RAN is congested and/or when the UE's RF conditions are poor, as shown in Fig. 2.12. As a result, the per-packet RLC latency and subsequently, the TCP RTT increase, adversely impacting the growth of the TCP `cwnd` during an AR burst, deteriorating the end-to-end performance of the AR session. While using smaller IP packets can help address this issue, they increase network overhead due to generation of a higher number of packets for the same burst and under-utilization of the available RAN capacity. With sub-optimal, smaller IP packets, this overhead becomes significantly high, affecting application goodput (see Fig. 2.12d). We propose a technique to optimize the packet sizes of the AR app by heuristically addressing this non-linear trade-off, based on underlying RAN conditions. In particular, when there is significant RLC segmentation of the packets, we reduce the IP packet size closer to a moving average of the instantaneous RLC PDU sizes for the UE. We carefully adapt IP packet sizes so that the gain from a quicker increase in the TCP congestion window for an AR burst offsets the loss from additional overhead of using more IP packets for the same burst.

Packet sizes can be varied by configuring the Maximum Segment Size (MSS) of the AR flow or Maximum Transmission Unit (MTU) of the AR UE. We conduct CloudAnchor experiments by adapting the IP packet sizes of the AR streaming session over public LTE networks under different network conditions (both congested and less-congested scenarios) using our technique and present the results in Fig. 2.12a. We evaluate the packet sizes selected by our technique (650 bytes) against the default large packet size of 1430 bytes and a smaller sub-optimal packet size of 400 bytes.

In Fig. 2.12a, for a more congested public LTE network (*i.e.,* campus) scenario, the default large packet size of 1430 bytes undergoes significant segmentation (around 7 RLC PDUs per packet) and hence, the aggregate RAN latency is high ($\sim$11 s). The optimal packet size for this scenario, yielded by our technique, is a smaller value, around 650 bytes. Upon setting this, the aggregate RAN latency is reduced by 37% and 58%, when compared to 1430 bytes and 400 bytes, respectively. At the same time, the network throughput and the AR application goodput from a 650-byte packet size increases by over 62% and 150% than the 1430-byte and 400-byte packet sizes, respectively (Fig. 2.12a, 2.12b). The 400-byte packet sizes achieve lowered throughput despite similar RLC segmentation to 650-byte is because the former under-utilizes the network capacity, observed by the maximum TCP `cwnd` for 400-byte reaching only 493KB, while 650-byte and 1430-byte can reach 657KB and 690KB, respectively. However, in low network congestion environments (*i.e.,* the mall), reducing the packet size has little impact on aggregate RAN latency because the eNB already allocates a larger RLC PDU to the device, resulting in little RLC segmentation. Hence, our technique selects the default large packet size close to 1430 byes. In conclusion, network-aware AR app design by choosing smart packet sizes for the app can improve aggregate RAN latency, end-to-end AR latency, network throughput and application goodput.

2.5.2 Network-Agnostic Optimization: Small Background Traffic

Another AR design optimization we propose is “priming” the eNB with information about the amount of data that the AR application will transfer in its next burst. This technique is network-agnostic, without the need to adapt to variations in the RAN. Typically, when the hosting device starts sending either a new uplink AR burst or new data in the middle of an AR burst after a longer idle period (lasting for tens of milliseconds), the UE has to request for resources from the eNB, incurring protocol signaling latency. The eNB is initially unaware of the uplink sending buffer, and may only allocate a small uplink grant (max. 125 bytes) for the UE. Then, upon data PDU transmission, the UE also piggybacks the uplink buffer size, which the eNB subsequently uses to allocate larger resource grants. This causes RLC segmentation, increasing the per-packet RAN latency, especially in congested scenarios. In order to make the eNB aware of the device's uplink buffer during an AR session, we generate small amounts of background uplink traffic, using an `icmp` packet of 100 bytes every $2-5$ms. Since there is an active small data transfer even during inter- or intra-AR burst idle periods, the UE is always scheduled minimal resources and it constantly piggybacks information about its uplink sending buffer to the eNB. This maximizes buffer-aware scheduling for the UE, which minimizes protocol signaling latency and RLC segmentation. In Fig. 2.12c, we plot the aggregate RAN latency and amount of outgoing data, with and without the additional background traffic, for an AR session over public LTE network. The results show that this small background traffic helps reduce aggregate RAN latency by ~50% on average, at the cost of a negligible increase in outgoing data size (including the extra background traffic). Our UE logs show that the average uplink resource grant for each MAC PDU during the AR session increases from 593 bytes to 1191 bytes with small background traffic.

2.5.3 Discussion: Application-layer Optimizations

Finally, we briefly discuss other potential application-layer solutions to reduce AR latency. In our existing experimental setup, the network data transmissions were opaque due to the internals of the Google ARCore platform being closed source. However, we hypothesize that the data transmissions consist of device data that is used for localization, as localization is known to be an integral part of AR [101] Reducing the fidelity of the device localization data, for example by quantizing the data or sub-sampling the data in time, could reduce the amount of data requiring transmission and thus the network latency. On the other hand, this may reduce device localization accuracy and impact the placement of virtual objects in the user's display; thus, we intend to explore such effects in future work, using open-source AR systems [70] that allow modification of the application layer.

2.6 Conclusions

The goal of high quality AR has engendered tremendous amount of research, but there has thus far been little focus on the impact of the cellular network. In this chapter, we show through extensive measurements on both an industry LTE testbed and public LTE that RAN latency is a significant part of the end-to-end AR experience, accounting for nearly 31.2% of the total latency. Unless this is reduced significantly, there is little hope for achieving AR with high QoE. However, our results also provide hope: AR traffic is very bursty in nature, making it a suitable candidate for practical traffic management schemes like QCI (which improves latency by up to 33%). Further, we also design network-aware and network-agnostic optimizations that improve latency by ∼40-70%. Future work includes a longitudinal study of AR users to learn specific AR app behaviors, which can then drive the development of a smart QCI-based scheduler specifically tailored for AR traffic characteristics. We will also quantify how 5G technologies can help close the gap of achieving seamless multi-user AR QoE by reducing the overall RAN latency.

Chapter 3

Power Thrifty Object Detection and Tracking for Mobile Augmented Reality

3.1 Introduction

AR is popular in the market today [81] with potential applications in many fields including training, education, tourism, navigation, and entertainment, among others [28]. 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 [61, 31, 76]. 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 [84]. 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 [47, 76].

To track real objects, AR apps can use tracking by detection techniques [101], 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 [16] 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 [51]. 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., [95, 52]) 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 [54] or via model compression [50]); however, they focus on individual DNN executions on individual input images [55], 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 [127, 63], 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 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.

MARLIN addresses several challenges in the domain of energy-efficient AR: (1) It provides highly accurate object classification and dynamically tracks the changing locations of multiple different objects in the scene, in order to place the virtual overlays correctly. (2) It reduces CPU throttling in cases where object detection computation demands exceed the compute capability, since built-in CPU throttling can significantly worsen tracking performance; (3) It preserves accuracy while reducing energy in challenging environments such as occlusions and/or zooming which are likely when the AR camera is worn/held by a mobile user; specifically, it does not over-trigger DNNs in response to camera motion; and (4) MARLIN is software-based and does not need specialized hardware. Thus, it is compatible with most modern mobile platforms. MARLIN's software (executables) can be downloaded via the project website [10]. To the best of our knowledge, this is the first detailed design, implementation and evaluation of an energy-thrifty object detection and tracking software framework for mobile AR. Overall, our contributions are as follows:

- We develop a framework, MARLIN, to manage the energy usage of AR, by mediating between two different object tracking approaches: tracking by detection using DNNs and incremental tracking via lightweight methods. MARLIN balances between achieving good tracking accuracy and energy efficiency by triggering DNNs only when needed. The decreased computation demands of MARLIN also reduce instances of automatic CPU throttling and its negative consequences on system performance.

- Within MARLIN, we design a novel lightweight change detector to determine when to trigger DNN detection, with very low false positive rates (crucial for reducing energy usage). Our key idea is to only examine portions of the frame outside of currently tracked objects to determine if new objects are present, while also ignoring effects from camera motion and occlusions.

- We implement and evaluate MARLIN on Android smartphones, using both standard video datasets [64] and through live experiments. Our results show that MARLIN can save energy by up to 73.3%, while losing at most 7.36% accuracy for 75% of the cases as compared to Tiny YOLO, the best baseline periodic DNN-based tracking by detection method we found in our experiments. Surprisingly, we find that in 46.3% of the cases, MARLIN both saves energy and improves accuracy, a win-win situation, compared to this best baseline. This is because MARLIN uses temporal information to avoid triggering tracking by detection, when the scene is noisy and thus detection would likely yield wrong conclusions.

- MARLIN is designed as a general framework that can work with a developer's chosen DNN, with or without a mobile GPU, and still save energy. To illustrate this, we incorporate multiple different DNN models (Tiny YOLO [95], MobileNets [100], MobileNets using mobile GPU [112], and quantized MobileNets [57]) into MARLIN's framework, and show that across these models, MARLIN can save energy by 45.1% while losing 8.3% accuracy, on average (compared to baselines of continuous DNN executions).

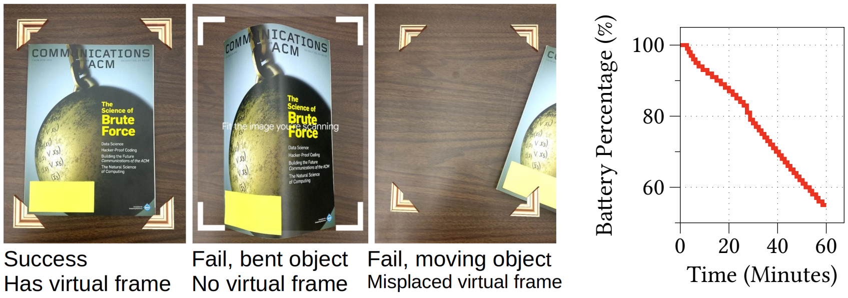
3.2 Motivation

The need for DNNs in emerging AR applications: AR systems are capable of understanding the 3D geometry of the scene (e.g., using simultaneous localization and mapping), but object detection is needed in AR to determine the locations of the virtual annotations in the first place [76, 61, 31]. Current AR systems used in practice are only capable of identifying surfaces or detecting specific instances of objects. For example, the open-source ARToolKit library [16] is designed to track specific fiducial markers placed in the scene (e.g., a QR code), while Google ARCore and Apple ARKit [44, 15] can detect flat surfaces or specific instances of flat objects (e.g., a specific magazine cover, but not the general class of magazines). These object detection capabilities are insufficient for AR applications such as public safety, where general classes of potentially moving, non-flat objects must be detected and recognized with high accuracy (e.g., moving victims needing rescue).

To demonstrate this, we experimented with a demo ARCore app [45] to detect objects of interest (Fig. 3.1a). We supplied ARCore with an image of a magazine for its internal training. At test time, ARCore was only able to detect the magazine under certain conditions: if the magazine was flat and non-moving. Based on our understanding of the code (full details are unknown because the code is closed source), we hypothesize that this is because ARCore only searches the camera frame for affine transformations training set items (i.e., translation, scaling, shearing, or rotations), and only when the scene is static - a bent object represents a non-affine transformation from a training image, and thus, detection fails.

Such poor or inaccurate detection/classification could result in missing or misplaced virtual overlays, potentially obscuring key portions of the scene and/or confusing the user. Therefore, we argue that the use of state-of-the-art DNNs, which consistently win the ImageNet object detection competition [99], is apt in order to correctly locate and classify the objects in the scene. DNNs are capable of detecting general categories of objects (e.g., human, animal, vehicles) under a variety of conditions, even if that specific object has never been seen before in the training set. For example, later in §3.5.3, we show that our DNN-based prototype can successfully detect people with high precision, even though we never used their specific images to train the DNN. Compared to classical SIFT features and other machine learning methods from the AR literature [58, 117, 124], DNNs provide more than 2× accuracy improvements [126].

Unfortunately, a naive approach of plugging in DNN object detectors into current AR systems is likely to lead to poor performance due to the uninformed patterns of DNN executions. For example, ARToolKit runs object detection as often as possible (i.e., tracking by detection). Modifying its object detector to call a DNN would result in high energy expenditure due to almost continuous executions. This is true even when using relatively lightweight, compressed DNNs (e.g., Tiny YOLO [95]) optimized for mobile devices (more



(a) ARCore [44] object detection fails for non-affine transformations.

(b) Frequent DNN executions drain battery.

Figure 3.1: Detection with ARCore; Energy drain with DNNs.

details later). On the other hand, ARCore and ARKit, to the best of our understanding (the details are closed-source), only record the initial pinned location of an object from when it is first detected, and cannot incrementally track objects while they are moving [45]. Modifying ARCore/ARKit to call a DNN (which may not even be possible due to their closed-source nature) may improve the initial placement of the virtual overlay, but the overlay may not be able to follow moving objects. In our evaluation (specifically Fig. 3.6 in § 3.5.2), we show that executing an object detector only once at the beginning of tracking leads to low accuracy.

Energy costs due to frequent DNN executions: To ensure high object detection and tracking accuracy, a naive method is to execute DNNs as often as possible, as is done in several prior works [92, 54, 95]. To showcase the energy drain of such an approach, we tested state-of-the-art object detection and tracking on Google TensorFlow, one of the most popular machine learning platforms. We use a popular object detector for mobile devices, Tiny YOLO [95], which applies DNNs as often as possible to maximize the tracking accuracy. This can be expected to result in a rapid depletion of the smartphone battery. To showcase this effect, we perform experiments on a Google Pixel 2, the results of which are shown in Fig. 3.1b. The rapid energy drain is due to the nature of DNNs, which can contain tens to hundreds of computationally-intensive layers. Furthermore, executing the same model on another recent phone (LG G6) caused the CPU to throttle its duty cycle after the first few minutes of a video, resulting in a significant drop in tracking accuracy (details in §4.6). We also tested MobileNets on Tensorflow Lite, MobileNets with mobile GPU and quantized MobileNets, and found that this quick battery depletion due to frequent DNN invocations holds true regardless of models or GPU offloading (discussed in §4.6).

Given the above discussion, we argue that a key gap in realizing object detection and tracking on mobile devices is the lack of a powerful, adaptive, and intelligent framework, designed with the resource limitations on the phone (battery, CPU) in mind. Such a framework should try to achieve a good trade-off between tracking accuracy and energy efficiency. We design and implement such a framework, MARLIN, which is described in the following section. In Table 4.2, we compare the characteristics of MARLIN with that of other recent AR systems (details in §4.7).

System **Features**	**Liu et al. [76]**	**Over Lay [58]**	**ARCore [44]**	**Glimpse [83]**	**Deep Mon [55]**	**Tiny YOLO (Default-DNN) [95]**	**MARLIN**
Energy efficient				✓	✓		✓
No specialized hardware	✓	✓	✓			✓	✓
No offloading			✓		✓	✓	✓
Real-time updates	✓		✓	✓			✓
Copes with CPU throttling							✓
Uses DNN	✓				✓	✓	✓
Localizes objects	✓		✓		✓	✓	✓

Table 3.1: Comparison of MARLIN and related work

3.3 The MARLIN Framework

MARLIN's design is predicated upon the following goals:

- **Low energy:** First, targeted for battery-constrained mobile devices, MARLIN must achieve object tracking with low energy. This not only prolongs battery life, but also saves energy for other AR functions not addressed here (e.g., localization [101]).

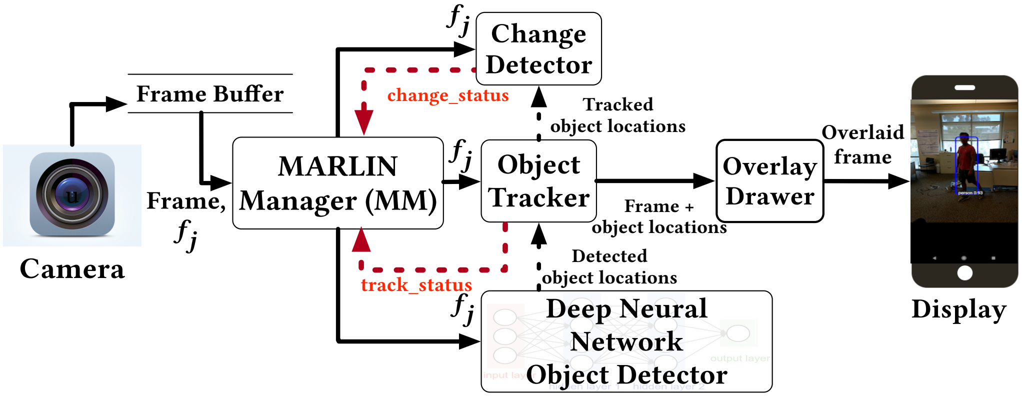


Figure 3.2: Overview of MARLIN's architecture

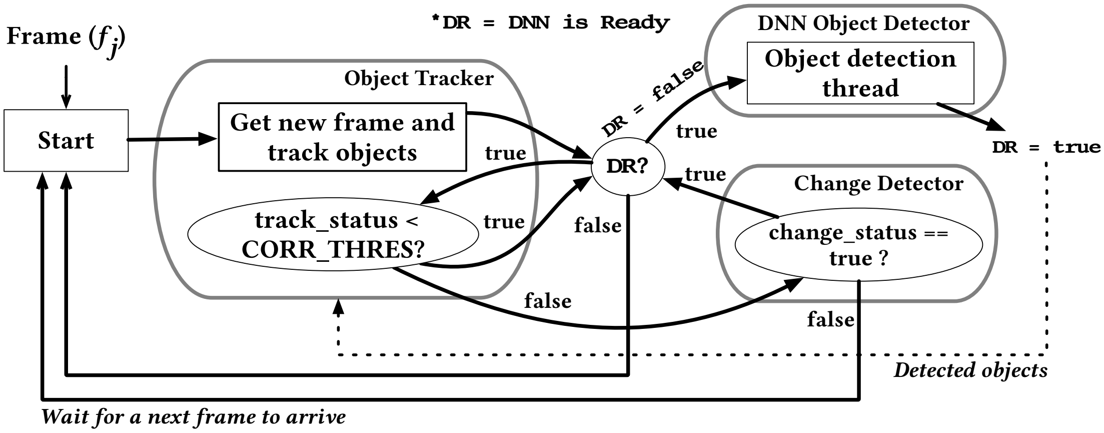


Figure 3.3: MARLIN Manager (MM)'s decision flow

- **Real-time performance:** Second, to enable very good AR experience, the detection and tracking of objects of interest must be done in near-real time, i.e., the location of each object must be updated frame to frame (within 33 ms for a 30 FPS camera).

- **Multiple accurate annotations:** Third, since we seek to overlay virtual objects atop the real world, the categories of (multiple) real world objects must be classified and their locations must be determined with high precision.

3.3.1 System Overview

Fig. 3.2 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. From this point, we abbreviate MARLIN Manager as ***MM***. MM is a real-time scheduler that assigns each incoming frame to one or more of MARLIN's three modules viz., the object tracker, the change detector, and the DNN object detector. These modules act as workers for MM, i.e., each module only processes frames that are assigned to it by MM.

By default, MM assigns a new frame to the *object tracker*, which updates the locations of the objects from one frame to the next. It returns a "track status" which indicates the fidelity of tracking and alerts MM of any changes to the current set of tracked objects, and triggers a new DNN execution if needed.

In addition, to check for new objects in a scene (that require tracking), MM assigns an input frame to the *change detector* module. While many change detection methods exist in the literature (e.g., [5, 129, 128]), we found experimentally that these approaches are unsuitable because they detect changes on both existing and new objects in the scene, resulting in high false positive rates and many unnecessary DNN executions (the main causes being changes due to camera movement or minor changes to the objects being tracked). To tackle this, we designed a new change detector that ignores objects that are already tracked with high accuracy by the object tracker, and only analyzes the portions of the frame that are “external” to the current set of tracked objects. The change detector issues an alert to MM if there are significant changes in these parts.

MM only sends a frame to the *DNN object detector* if it needs to detect/classify new objects in that frame, or when features of the currently tracked objects change significantly and need to be detected anew. This is because the DNN is MARLIN’s most energy-draining module and should only be invoked on a need-to basis. MM uses tracking information and the change detection in a principled way to decide if the frame should be assigned to the DNN.

Finally, the object tracker conveys 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.

3.3.2 MARLIN Manager (MM)

In this subsection, we describe MARLIN Manager or MM in greater detail. At a high level, the logic embedded in MM employs the lightweight change detector and object tracker modules as often as possible, and triggers the DNN only if either of these modules indicates that a significant change has occurred in a frame (compared to a prior frame). It uses a “short-circuit OR” decision flow that only runs the change detector if the object tracker did not trigger a DNN, thus avoiding wasted computation/energy.

Functional description: Fig. 3.3 depicts the decision flow executed by MM . MM obtains input frames from the camera in the form of a byte array with dimensions specified by the three color channels (red, green, blue), and 640×480 pixels (down-sampled from the original resolution, and configurable by the user). Each such frame is assigned to the object tracker. MM waits until the object tracker updates the locations of the objects of interest and returns the correlation between the tracked objects in the current frame and in a previous frame (the returned correlation value is referred to as track_status). This correlation captures the fidelity of the tracking across frames (details in §3.3.3). If track_status is less than a threshold (CORR_THRES), MM attempts to trigger the DNN. Note here that CORR_THRES depends on the desired fidelity of tracking. If higher fidelity tracking is needed, smaller changes (a lower threshold) will need to trigger the DNN (causing these to be more frequent at the cost of higher energy); a lower acceptable fidelity translates to a higher threshold.

If the track_status is higher than CORR_THRES (meaning that there were no significant changes in tracked objects), then the second operand in the short-circuit OR needs to be evaluated, and so MM starts a change detector thread. This checks if there are changes in the background that could also require the invocation of a DNN. Upon completion, the change detector returns a value (called change_status) that indicates whether a significant change in the current frame relative to the immediately preceding frame was detected (details in §3.3.4). If a significant change is indicated, MM initiates an invocation of the DNN.

In order to prevent repeated DNN invocations due to dynamic changes (e.g., the correlation could be lower for several successive frames), MM checks if or not a DNN invocation has already been made in the immediate past by checking a flag variable, DR (for DNN is Ready). If a DNN thread is already being executed, the flag DR will be false and MM will simply abort the DNN execution attempt. Whenever a DNN is invoked, MM marks the flag DR as false in order to block other frame assignments to the DNN. Essentially, the DR flag ensures that there is only one running DNN thread at any given time, in order to prevent repetitive invocations and thereby ensure that the CPU does not get overloaded or throttled.

Exceptions: If MM cannot finish all the above operations before a new frame arrives, a frame in the buffer is overwritten by a new one. If the change detector thread takes more than one frame to finish (and thus does not return a value within a frame), MM will trigger the DNN at that later time. These exceptions are very rarely observed in our experiments, and even when observed, the delay (2-3 frames) does not affect user experience (not noticeable). If there are no objects being tracked by the object tracker, the tracker returns a zero correlation value, causing a DNN invocation.

3.3.3 Real-time object tracker

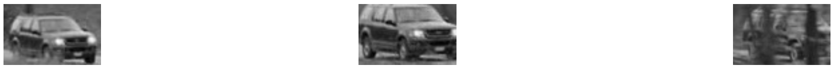
MARLIN needs to continuously track objects of interest (detected by the DNN module) across successive frames, as the object moves/morphs in the scene. To conserve energy, MARLIN's object tracker needs to use (a) very lightweight feature extractors and (b) very lightweight object tracking algorithms. To assess the tracker's performance as it runs, we need some metric that can be computed online; the metric should be able to readily provide a means of determining when the tracking quality has degraded and a new DNN execution is needed (to fully refresh the object locations). We discuss these design considerations and how they influence object tracker design.

Feature extraction: We examined popular feature extractors in the literature. While SIFT features have been used in previous AR systems [58, 59, 124], we chose to use

ORB (Oriented FAST and Rotated BRIEF) features in the tracker because they can be extracted in near real-time even on smartphones. ORB has been shown to be 14× and 341× faster than SURF and SIFT respectively with very good tracking precision [98, 121], and we have experimentally verified that extracting SURF/SIFT features for even a single object in a frame takes hundreds of milliseconds, while our object tracker, including ORB feature extraction, takes less than 10 ms (see §3.5.2).

Object tracking: While heavyweight DNN-based object trackers can provide good tracking accuracy (e.g., [56]), these are unsuitable for mobile devices due to their expensive computation of multiple DNN layers. Our goal here is to estimate the *optical flow* of features, which captures the pattern of motion of objects between successive frames. Instead of trying to design a method from scratch, we use the well-known Lucas-Kanade method [17]. This method estimates the local image flow (velocity) vector (V_x, V_y) using keypoints (features) in the window (in this case the object position box to be tracked) and assumes that these keypoints should move together with this velocity. It has m equations (m keypoints) to solve for two unknowns V_x and V_y, using a least-squares approximation [79]. It makes three assumptions viz., brightness constancy (the same keypoint appearing in both images should look similar), limited motion (keypoints do move very far), and spatial coherence (keypoints move within a small neighborhood) [79]. This method has been shown to be well suited for object tracking [33], and our experiments show that it is also energy-efficient (see Fig. 3.6 of §3.5.2).

One important parameter is the neighborhood size that the Lucas-Kanade method searches to find matching features. If the neighborhood size is too small, the object tracker



(a) Frame 1 (b) Frame 2 (c) Frame 3

Figure 3.4: Cross-correlation decreases from 0.92 (frame 1→2) to 0.69 (frame 2→3) due to occlusion.

cannot track fast-moving objects accurately. If this neighborhood size is too large, the tracking latency becomes too large because of the larger sample space that needs to be examined for feature matching. We empirically tested this parameter on different videos, measuring the latency and CPU resources utilized for tracking, and found a size of 7 to yield both good accuracy and acceptable latency. A neighborhood size of 7 means that for each feature, the Lucas-Kanade method scans all the features in a 15 × 15 pixel area to find a matched feature (a center pixel plus 7 pixels above, below, left, and right).

Metric for tracking accuracy: Unfortunately, tracking is not always accurate with respect to changes in object locations. To have a perfect metric to quantify accuracy, we would require the ground truth information about object locations, but this is impossible to have in a real-time, online system. Therefore, in MARLIN, we choose to measure the accuracy of the tracker using the normalized cross-correlation (NCC), which is a well-known technique for template matching [123]. NCC provides a measure of the similarity between two images and is given by: $NCC(f,g) = \frac{1}{|R|}\sum_{i,j\in R} f(i,j) \cdot g(i,j)$ where, f and g are the two images, R is their (bounding box) area, and i, j are the pixel locations within the images.

Example: Fig. 3.4 depicts the car in frame 1 to be traced to find its new location in frame 2. The object tracker calculates the NCC between the two boxes by using the above equation, and finds the correlation value is 0.92. Next, the car is tracked from frame

2 to frame 3; the correlation is 0.69 (frame 3 has occluding trees), because of a moderate accuracy drop (i.e., the tracked object is 69% similar to that in the previous frame).

We use a default correlation threshold of 0.3 to trigger the DNN; we consider that if the similarity is less than 30%, the object must be lost (the DNN helps detect objects and recovers accurate locations again). Note that for AR, we need a reasonable level of correlation with respect to the location of a classified object, and "perfect" correlation is not needed. A more stringent threshold (e.g., 0.5) will cause more frequent DNN invocations and thus higher energy. As shown in §4.6, our default threshold yields good accuracy.

Runtime execution: Putting all of these components together, the object tracker functions as follows. The input to the object tracker is the current frame, and a list of tuples (`objectID`, `class Label`, `objectLocation`, `detectionConfidence`) containing information about the detected objects. `objectID` is a unique number associated with each detected object, `classLabel` is the class to which the DNN attributes the object (e.g., tiger), `objectLocation` is a 4-tuple vector (left, top, width, height) representing the location of a detected object, and the confidence of the DNN in making the classification decision is given by `detectionConfidence`$\in \{0, 1\}$.

For each detected object, the object tracker executes the following steps: **(i)** For the detected object location in frame j (where j is the most recent frame number seen by object tracker or DNN execution), extract the ORB features F_j (keypoints); **(ii)** For the current frame $j+i$ (i is the number of frames since the last DNN or object tracker execution), extract the ORB features F_{j+i} in the neighborhood of the detected object location from the previous step. **(iii)** Use the Lucas-Kanade method to estimate the optical flow from F_j to

F_{j+i} and estimate a new rectangular box that covers the matching features. This new box is the updated location of the object. **(iv)** Compute the minimum NCC (across all objects) between the updated and previous locations (`track_status`) and pass this to MM, which triggers a DNN execution if this NCC is below a threshold.

3.3.4 Lightweight 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); such changes would also trigger the DNN. 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 [5, 129, 128] 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 MM, 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 (example in Fig. 3.11); **(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 [34]).

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, MARLIN 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 (details in §3.5.1). 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

Layer	Filter	Size	Stride	# Params	Layer	Filter	Size	Stride	# Params
c_1	**16**	3×3	**1**	**448**	c_5	**256**	3×3	**1**	**295,168**
m_1		2×2	**2**		m_5		2×2	**2**	
c_2	**32**	3×3	**1**	**4,640**	c_6	**512**	3×3	**1**	**1,180,160**
m_2		2×2	**2**		m_6		2×2	**2**	
c_3	**64**	3×3	**1**	**18,496**	c_7	**1024**	3×3	**1**	**4,719,616**
m_3		2×2	**2**		c_8	**1024**	3×3	**1**	**9,438,208**
c_4	**128**	3×3	**1**	**73,856**	c_9	**175**	1×1	**1**	**179,375**
m_4		2×2	**2**		r				

Table 3.2: MARLIN's DNN architecture (based on [95]).

classifiers such as Support Vector Machines, but found experimentally that random forest had the highest change detection accuracy.

Runtime execution: MM 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 MM to trigger the DNN (if needed) as soon as the result is received.

3.3.5 DNN based Object Detector

Next, we briefly describe the DNN module within MARLIN.

Functional description: The input frame received by the DNN module from MM is passed through 16 layers (using the `recognize Image()` method of Tensorflow) sequentially as shown in Table 3.2, where $c_i, i \in \{1, 9\}$ represents a convolutional layer, $m_k, k \in \{1, 6\}$ is a maxpooling layer, and r is a region layer which outputs the final prediction results containing object locations, class labels, and confidence values. The output of c_9 has a dimension of `gridWidth` × `gridHeight` × `boxes` × (`classes` + 5), where `gridWidth` and `gridHeight` are grid dimensions corresponding to the input frame, `boxes` is the number of prediction candidate boxes for each grid cell and `classes` is a list of class probabilities (a value for each class) with respect to object classification. The additional 5 dimensions represent the "objectness" of the predicted box (i.e., the probability that the box contains an object) and the box location `(x,y,w,h)`.

At layer r, a softmax function [21] outputs the confidence that an object belongs to a class. The confidence is computed as `confidence` = `objectness` × `class_prob`, where `class_prob` is the maximum value from the list of probabilities of belonging to the various classes. If for a given prediction candidate box, `confidence` is less than a threshold, that prediction box is ignored. In our evaluations, we set this threshold as 0.25 because this means that a box will be accepted if `objectness` and `class_prob` are both greater than 0.5. We have empirically found that this threshold yields a reasonable balance between object plausibility and the number of objects detected.

In summary, for each prediction box, the DNN predicts a center point, width, and height of an object, and how likely it is that the box contains an object (`objectness`). It finally outputs the class to which the object in the box most likely belongs (`class_prob`). Tiny YOLO computes these via a single pass through the network (from the image to the prediction), making it one of fastest DNNs for object detection on mobile platforms (latencies of state-of-the-art DNNs are compared in [96]). We also evaluate other possible DNN model choices in §3.5.2. Note that MARLIN executes pre-trained DNNs for real-time inference, with training being performed offline without power constraints (training details provided in §3.5.1).

Exceptions: If the DNN takes too long to complete, the object tracker has to track incrementally. It is possible that between the time that the DNN receives an input frame i and returns a result in frame $i+j$, there is a significant temporal distance, resulting in the object tracker failing to find the objects in frame $i+j$ detected by the DNN in frame i. If this happens, MM will invoke the DNN module again until tracking by detection succeeds in finding objects.

3.4 Implementation

We next briefly describe MARLIN's implementation, which realizes seamless interactions between multiple Android classes/threads.

Platform: We implement MARLIN on Android phones (LG G6 and Google Pixel 2 running Android 7.0 and 8.0, respectively). We use the TensorFlow [110] and OpenCV libraries [23] to implement the DNN and image manipulation functionalities, respectively.

Module implementation: MM runs within a CameraActivity class that extends Activity, the main UI class in Android. It starts when the MARLIN app is invoked by the user. A new frame is buffered in a byte-array in shared memory and MM fetches it once the memory has been written (subsequently the frame is dispensed to the other modules). ***Object Tracker*** is an instance of the class MultiBoxTracker, and provides methods for other components that want to exchange shared information. It runs in the main thread because it is fast (6-10 ms per frame with multiple objects) and does not block the UI. ***Change Detector*** is a background thread that copies a new frame from MM and calls `getTrackedBoxes()` of the object tracker to get the set of currently tracked objects; it also runs the algorithm in §3.3.4 to detect changes. ***DNN*** is also implemented as a background thread. A DNN thread can be interrupted and can save its intermediate results for further processing when it resumes. This allows the main UI thread to have access to the CPU even when a DNN thread is being run (so that the app is responsive to the user at all times). ***Overlay Drawer*** is a callback thread of the OverlayView Android class and fetches a list of tracked objects from the object tracker and draws them on the frame.

Information sharing: We use methods to pass parameters to/from the object tracker and use shared memory to communicate for real time operations. MM copies a frame to the working threads (change detector or DNN) only if it decides to call one of them.

Frame Synchronization: We use frame sequence numbers to ensure that the different components are synchronized with respect to frames. MM increases the frame sequence number by 1 for each new frame and is the only entity that can update this number.

Logging: MARLIN is instrumented to log CPU frequency, CPU temperature, locations of tracked objects in the scene, and the latency of each component of MARLIN. ***Object location:*** In the object tracker code, we log frame identifiers, object locations, and class labels into storage, and use these logs to compute the accuracy offline. ***Energy:*** Since the phones do not provide direct physical access to the battery, we use software tools to measure energy consumption. On the LG G6, we use Qualcomm's Trepn Power Profiler app [90], and on the Google Pixel 2, we use Android system logs (due to Trepn's lack of support for the Google Pixel 2). Specifically, we read the Android virtual files `current_now` and `voltage_now` from the `/sys/class/power_supply/battery/` directory to obtain current and voltage (used to compute power). The battery level values are read from the `ACTION_BATTERY_CHANGED` Android system variable. ***CPU:*** We read the CPU frequency and temperature from the virtual files `scaling_cur_freq` and `thermal_zone10/temp` every 200 ms. The CPU load is then estimated as $\frac{cpu\_freq}{maximum\_freq} \times 100$. We estimate these metrics because recent Android versions since Marshmallow adjust CPU frequencies in response to load (here mainly DNN executions) in real-time [42].

3.5 Evaluations

In this section, we describe the experimental evaluations of MARLIN. We first provide brief discussions on details such as our training and test sets and the metrics for evaluations.

3.5.1 Prerequisites and Metrics

Baselines, Model Training and Inference: We first describe the baselines used for comparisons and the training and test datasets that we use.

Baselines: We consider five different DNN models and perform continuous invocations of these as our baseline cases; we also consider a subset of these models as appropriate as the DNN object detector in MARLIN. The five models are abbreviated as follows: (a) **YOLO** [95], which is a 30-layer DNN detector that provides high accuracy on servers but is typically not used in mobile systems because of its high power consumption and latency; we consider it for completeness but do not use it as an object detector in MARLIN. (b) Tiny YOLO or **TYL**, which is a compressed 16-layer version of YOLO. (c) MobileNets [100] or **MNet**, which is trained and run on the Tensorflow Lite [113] framework. Tensorflow Lite is TensorFlow's lightweight platform for mobile and embedded devices; this provides us with insights with regards to MARLIN's energy savings capabilities on an already optimized mobile software platform. (d) MobileNets using mobile GPU or **MNet-GPU**, which offloads expensive computations to a GPU for low power [55, 66]. (e) MobileNets quantized model or **MNet-Q**, which quantizes the DNN weights in order to reduce execution latencies, and possibly also the DNN execution energy [50, 57].

In terms of notation, when we consider the continuous invocations of one of these DNN models, we include the prefix "Baseline" (e.g. Baseline-TYL). When we use a DNN model as the object detector in MARLIN, we apply the prefix "MARLIN" (e.g. MARLIN-TYL). Because we experimentally find that Tiny YOLO has the best accuracy compared to the other models, we later consider it both as the baseline and as the object detector in MARLIN;

thus, we subsequently refer to "Baseline-TYL" as "Default-DNN" and to "MARLIN-TYL" as "MARLIN". Further details are provided in §3.5.2.

We also compare MARLIN with handcrafted approaches that invoke the Tiny YOLO DNN after skipping a fixed (K) number of frames; the extreme case is when $K = \infty$; *i.e.,* when incremental tracking is used continuously after the initial detection, which we call **Inc. Track**. Our baselines are inspired by similar approaches from the literature (*e.g.,* continuous DNN invocations [92, 55], incremental tracking [101], periodic DNN executions [127]).

Model Training and Inference: In this section, we describe our machine learning model training and testing methodologies.

DNN model training: We train these models with the ImageNet video dataset [99], consisting of 3,862 video clips (1.1 million frames) containing 30 categories of objects, with ground truth labels provided. We split the dataset and use 95% for training and 5% for validation. We calculate model accuracy on the validation set every ten training epochs to check if the model was overfit (accuracy starts to fall). For YOLO models, we adjust learning rates relative to training epochs as specified in [95], and for MobileNets models we use learning rates specified in the default training scripts [111].

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 (details in §3.3.4) 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.

Model inference: After training the models offline on a server, we load them on Android phones with the appropriate TensorFlow and OpenCV libraries. While we evaluate the system performance using accuracy and energy metrics (details upcoming), DNN inferences are called by MM. Note that neither the DNN models nor change detector models see the test videos during training time.

Metrics: We evaluate MARLIN 's accuracy in classification and tracking and its energy consumption.

Accuracy metrics: To quantify the accuracy of classification and tracking we use the following metrics [27, 119]:

- *Average Classification Precision (ACP):* Given frame i, we compare the predicted class labels with ground truth labels and count all the matches as *true positives (TP)*. We count unmatched labels as *false positives (FP)*. Then, the ACP of frame i is $ACP_i = \frac{TP}{TP+FP}$. The ACP of a video is computed as the average ACP of its frames.

- *Average Intersection Over Union (IOU):* If the predicted class label of an object matches a ground truth label, we calculate the IOU as the overlap between the predicted and ground truth regions. We perform dataset experiments where we use the provided ground truth data; we also do live experiments where we use a powerful object detection method, viz., YOLO (details in §3.5.3) as the ground truth. The IOU of object j in frame i is $IOU_j^i = \frac{R_j^G \cap R_j^P}{R_j^G \cup R_j^P}$, where R_j^G is the ground truth region of object j, and R_j^P is the predicted region of object j. We average the IOU for all the predictions per frame, and finally average the IOU across all frames in the video.

We point out that even the state-of-the-art object trackers achieve at best a 65% location accuracy [27] using the IOU metric (for example, a 65% IOU corresponds to 79% of the predicted region overlapping with the ground truth region, if both regions have the same area, using the equation above). These accuracies suffice for the applications we have in mind; the relatively low accuracy only causes small displacements of the real-world objects, and thus does not majorly affect the placement of augmented objects.

Energy metrics: We use power and battery life to evaluate the energy consumption of MARLIN. We log energy samples every 200 ms (as detailed in §4.5) and compute the average over the period of an experiment to compute power. To measure the energy of MARLIN's individual components, we successively enable each component and estimate the additional energy consumption as that component's power. For example, if we measure the OS plus screen as consuming 1000 mW, and then enable the camera and measure a total power of 2800 mW, we esimate the camera's power as 1800 mW. To compute battery life, we record the starting battery level (b_s) and the final battery level (b_f) in each experiment (according to §4.5). We then perform linear regression to estimate the total battery life as $BL = \frac{p \times 100}{(b_f - b_s) \times 60}$, where p is the duration (minutes) of each experiment.

3.5.2 Offline Dataset Experiments

First, we evaluate MARLIN's performance offline on a standard video dataset with known ground truth, across a diverse set of environments. Our complete dataset includes 80 test videos [64] with a variety of objects (*e.g.,* trains, animals, cars), single and multi-object scenes, and fast and slow-moving scenes, meant to emulate a variety of settings under which AR could be used. In each video, the number of objects varies between 1 and 15, and the average object motion between consecutive frames (the Euclidean distance between an object's center in frames i and $i + 1$) ranges from 0.5 to 10.7 pixels. Since the videos are relatively short (hundreds of frames), and we want to capture the effect of a longer AR experience within the same environment, we loop the videos to have a total duration of 10,000 frames per video. We allow a 5-minute cooldown period between each video to reset the phone's state.

To begin with, to keep the time duration of experiments within reason (given the limited number of phones at our disposal), we consider 15 videos and compare the performance of MARLIN with all the baselines and DNN models described earlier, as well as several handcrafted frame skip approaches. Each set of experiments with a given DNN

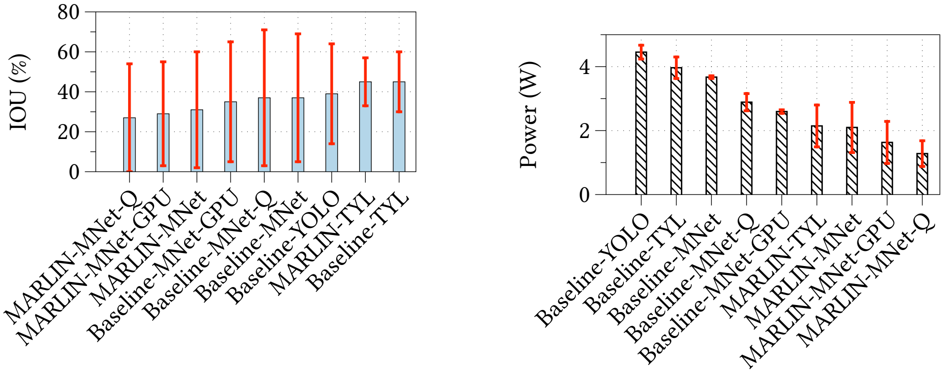


Figure 3.5: With four different DNN models, MARLIN saves 45.1% power while losing 8.3% IOU, on average.

takes three hours (running 15 videos, cool down, phone recharging). These experiments represent different types of object classes and various levels of motion. From § 3.5.2 we present experimental results with the entire set of 80 videos and compare the performance of MARLIN with the best found DNN (Tiny YOLO).

Comparison with the baseline approaches.

Compared to continuous executions of compressed DNNs that are optimized for mobile devices, MARLIN **reduces power by 45.1% while losing 8.3% IOU, on average.** We plot the average power and accuracy of the various approaches considered in terms of IOU in Fig. 3.5. First, we note that uncompressed YOLO consumes the most power due to its model complexity, but its average IOU over time is lower than Tiny YOLO (its compressed counterpart) due to its high detection latency (4500 ms vs. 1200 ms).

This is because when detection latency is high, YOLO fails to detect fast-moving objects (e.g., a landing airplane) in time. Therefore, we focus on compressed and optimized models such as Tiny YOLO and MobileNets.

Second, we note that continuous execution of MobileNets (Baseline-MNet) achieves lower IOU and consumes similar energy to continuous execution of Tiny YOLO (Baseline-TYL) [1]. Third, MARLIN with MobileNets (MARLIN-MNet) saves 42.8% power consumption with a 10.6% reduction in IOU, compared to a continuous execution of MobileNets (Baseline-MNet-GPU). Similar energy savings hold for MARLIN with Tiny YOLO (MARLIN-TYL vs. Baseline-TYL), and for MARLIN with quantized MobileNets (MARLIN-MNet-Q vs. Baseline-MNet-Q). Fourth, with regards to the MobileNets variants, (regular) MobileNets, quantized MobileNets, and MobileNets with GPU achieve similar accuracy; in terms of power, mobile GPU and model quantization save 29.3% and 21.3%, respectively (Baseline-MNet-GPU, Baseline-MNet-Q vs. Baseline-MNet). The key observation is that even though the use of the mobile GPU already saves 29.3% of power, MARLIN can further save an additional 37.1% (on top), with a hit of just 9.9% in terms of IOU (MARLIN-MNet-GPU vs. Baseline-MNet-GPU). Overall, these results suggest that MARLIN is a general framework that is useful across a variety of compressed DNN models, even with a mobile GPU. Because it exhibits the highest accuracy (and similar power consumption to other DNN models), we use Tiny YOLO as the default baseline (default-DNN) and as MARLIN's object detector in all subsequent experiments.

[1] The standard deviation of the IOU for MobileNets tends to be higher than that of Tiny YOLO because MobileNets sometimes misclassifies objects when they are small or blend in with the background, leading to low IOU. See §4.8 for further discussion.

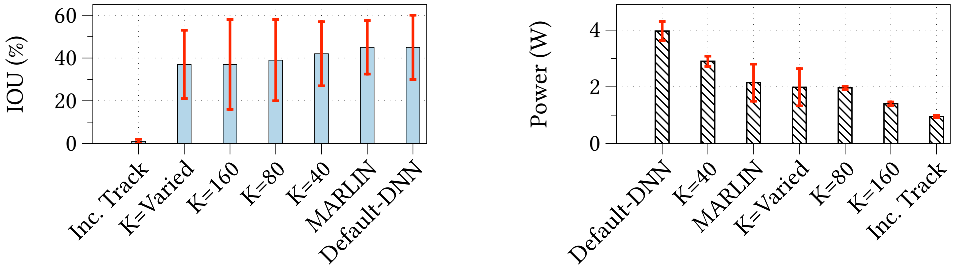
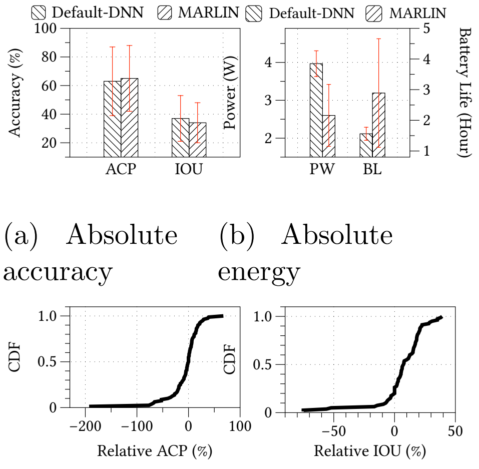


Figure 3.6: Compared to only tracking or periodic DNN executions, MARLIN has higher accuracy and/or lower energy.

Comparison with other hand-crafted approaches

MARLIN achieves 19× higher accuracy than the incremental tracking approach, and lower energy for the same accuracy compared to the best constant skip approach. We compare MARLIN against a constant skip approach (with different skip periodicity $K = 40, 80, 160$) and an incremental tracker baseline ("Inc. Track") in Fig. 3.6 for 15 different videos, where the average number of frames between DNN invocations by MARLIN ranged from 38 to 833. First, we see that "Inc. Track" suffers from very low accuracy compared to all other approaches (19× lower than MARLIN); this is because when the tracker loses track of objects, there is no recovery from object (re)detection available; thus, we do not consider this approach further. MARLIN achieves comparable IOU with the best constant skip approach ($K = 40$) but consumes 26% less power because it intelligently chooses to trigger fewer DNNs. Moreover, even if we "cheat" by hard-coding the value of K to the average value as chosen by MARLIN for each video ($K =$ Varied), the accuracy of MARLIN is still higher on average because MARLIN chooses when to invoke the DNN, as opposed to fixed periodic executions that ignore the scene content. Finally, default-DNN



(a) Absolute accuracy

(b) Absolute energy

(c) Relative ACP

(d) Relative IOU

Figure 3.7: MARLIN saves energy with mimimal accuracy degradation.

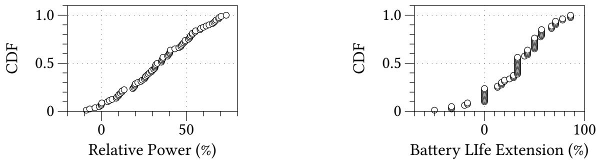


Figure 3.8: Per video, MARLIN uses less power (left) and extends battery life (right), relative to the "default-DNN".

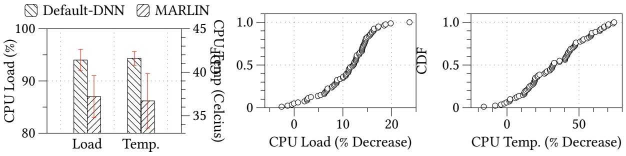


Figure 3.9: MARLIN reduces CPU load and temperature (left), relative to the "default-DNN" (center, right).

has the same high accuracy as MARLIN but consumes significantly more energy because it invokes additional unnecessary DNNs.

A closer look at energy and accuracy

MARLIN **extends the battery life by 1.85× on average with a small accuracy loss.** To see whether MARLIN can achieve good performance across a range of videos, we next evaluate the energy savings with MARLIN across a larger test set of 80 videos, and also examine the associated accuracy penalty compared to the default approach, which runs Tiny YOLO as often as possible. In Fig. 3.7a, we plot the mean and standard deviation of the ACP and IOU across all frames of all videos. For the same experimental runs, we plot the power and battery life in Fig. 3.7b.

	OS + Screen	**Camera**	**Object Tracker**	**Change Detector**	**DNN**
Power	**0.9 - 1.1**	**1.9 - 2**	**0.2 - 0.3**	**<0.1**	**1.7 - 1.9**
Latency	**-**	**-**	**8 ± 2**	**4 ± 1**	**1100 ± 100**

Table 3.3: Power (W) and latency (ms) of MARLIN's components.

These results show that MARLIN reduces power by up to 73.3% (34.5% on average), and extends battery life by 1.85×, with a small loss in accuracy ($< 10\%$). This is because MARLIN triggers tracking by detection significantly less often.

Beyond averages, we also compute the relative power per video as $\frac{p_d - p_p}{p_d}$, where p_d is default-DNN's power consumption and p_p is MARLIN's power consumption. Fig. 3.8 shows the CDF across videos, and we see that for 75% of the videos, MARLIN reduces power by at least 19% and extends battery life by at least 13%. Also, in 25% of the cases, MARLIN extends the battery life or reduces power by at least 50%. There are only 10% of cases wherein we do not see energy savings; a closer look reveals that these videos have very complex, high motion scenes; thus, DNN-based detection is necessary almost continuously, and MARLIN behaves similarly to default-DNN.

Finally, Table 3.3 shows a zoomed out view of the power and latency of each component of MARLIN. The results confirm that MARLIN's non-DNN components are lightweight, and focusing on the DNN executions which comprise a large portion of the total energy is key to reducing the overall power consumption.

For 75% of the videos, MARLIN **results in at most a 7.3% hit in ACP and a 18% hit in IOU.** To understand the performance of MARLIN further, we calculate the relative accuracy of object detection and tracking across videos when using MARLIN and default-DNN (calculation similar to relative energy). The CDFs of relative accuracy in terms of ACP and IOU, across the videos in the test set, are shown in Fig. 3.7c and 3.7d. For 75% of the videos, MARLIN results in a hit of $\leq$ 7.3% (ACP) and $\leq$ 18.0% (IOU). These modest drops show that MARLIN performs well while ensuring low power in tracking object locations and labels between frames. We note that approximately half of the tested videos are challenging due to fast motion or multiple objects, thus making this result very promising.

Surprisingly, for 46.3% of the videos, MARLIN **both achieves better ACP *and* consumes less energy.** We see from Fig. 3.7c and 3.7d that for a significant fraction of the test videos, MARLIN improves accuracy compared to default-DNN. A closer look indicates that for 46.3% of the videos, MARLIN both reduced energy and resulted in higher ACP compared to default-DNN. We find that these cases typically related to videos with a zooming or shaky camera. We will further discuss these special cases next in §3.5.2.

Sample Case Studies

We next present two sample case studies to provide an understanding of why MARLIN sometimes improves accuracy in addition to saving energy; other such cases typically relate to zoomed in frames, occlusions, or cluttered scenes where by using tracking or change detector features, MARLIN reduces DNN invocations that cause false positives/wrong detection.

In the case study of a zoomed-in video, MARLIN **has a 55% gain in ACP and saves 2,500 mW in power.** In this video, the camera is zoomed in on a

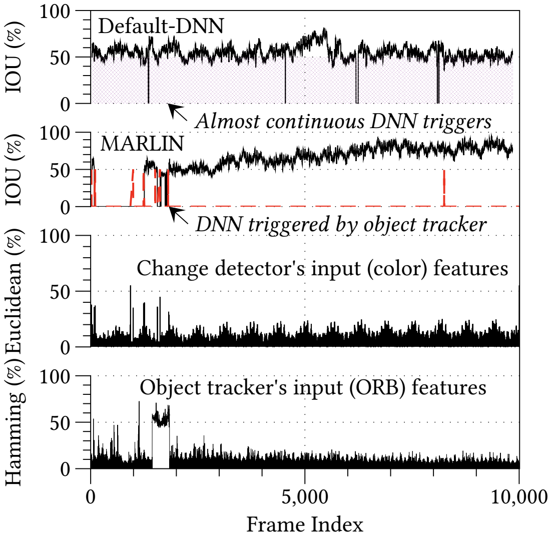


Figure 3.10: Case study. MARLIN achieves higher IOU using incremental tracking, rarely invoking DNNs due to the color/ORB features' stability.

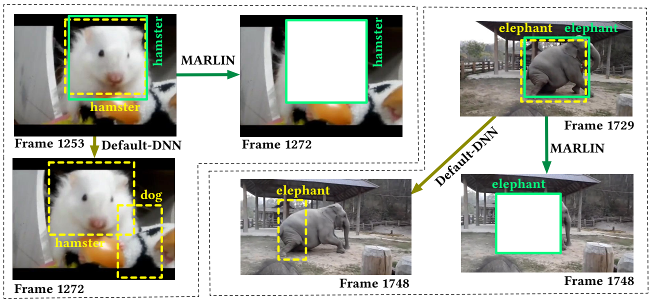


Figure 3.11: Sample frames of 2 case studies. MARLIN (solid green) is robust to small variations of currently tracked objects, while default-DNN (dashed yellow) re-triggers the DNN resulting in poor detection.

hamster. In the top two rows in Fig. 3.10, we plot the IOU over time for default-DNN and MARLIN. We see that default-DNN maintains a reasonable IOU by executing tracking by detection frequently (the dense vertical purple lines), while MARLIN actually *improves* IOU over time. This is because MARLIN 's incremental tracking and change detection use the manually-chosen ORB and color features that are stable over time. Thus, DNNs are hardly invoked. The stability of these features is seen in the bottom two plots in Fig. 3.10;

we show the Euclidean distances between color feature vectors across frames (used by the change detector) and the Hamming distances between ORB feature descriptors between consecutive frames (used by the object tracker).

In contrast, default-DNN chooses features automatically and frequently (with hidden convolutional layers), ignoring temporal correlation and causing the IOU to suffer[2]. More importantly, it yields false positives with respect to detected objects on many invocations. To illustrate this, consider Fig. 3.11. At frame 1253, both default-DNN and MARLIN detect the hamster correctly in the middle of the frame. The former then triggers the DNN again, which returns two objects in frame 1272: a hamster (true positive) and a dog (false positive) at the right bottom corner. MARLIN, however, continues to track the hamster found in frame 1253 and does not cause an erroneous DNN result in frame 1272. In frame 1272, MARLIN's precision is 100% while default-DNN's precision drops to 50%. We find that this effect repeats for this video and thus, while default-DNN only achieves an overall average ACP of 57% and an IOU of 54% with 400 DNN executions, MARLIN achieves an overall ACP of 87% and IOU of 69%, with only 12 DNN executions. This saves 2500 mW of power and extends the battery life by 3.5 hours.

In the case of a shaky video, MARLIN **improves the IOU by 52%.** An elephant is the focal point of this video, but it is sometimes occluded and suffers from the shaky motion of the camera. We find that only about half of the frames serve as good inputs to the DNN module. Both default-DNN and MARLIN have lower IOUs due to the challenging scene, but MARLIN achieves a 35% IOU while default-DNN only achieves 23%.

[2] DNNs that use temporal structure of videos have only been recently studied, e.g., for activity recognition [26] or object tracking [56], and are more complex/high energy [24].

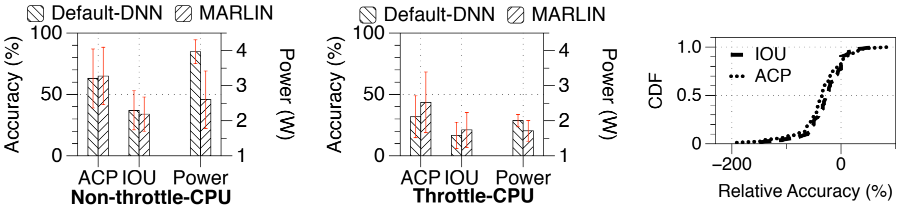
This is because MARLIN's incremental tracking ignores moderate noises in the scene (e.g., blurry/partially occluded frames), while default-DNN often performs DNN-based detection on such frames and captures poor object features for tracking. For example in Fig. 3.11, at frame 1729 with both methods, the DNN detects the elephant and outputs a box centered on the elephant and covering most of the body. However, at frame 1748, default-DNN triggers the DNN again but now the center of the elephant is falsely identified to be near the tail. This causes the prediction box to shrink, and the IOU is thus only 40%. MARLIN, on the other hand, does not trigger the DNN since its incremental tracking outputs a more accurate box with an 83% IOU, and the whiting out of the elephant also does not trigger the change detector.

Impacts on Mobile CPU

For 60% of the videos, MARLIN **reduces the load and temperature by 10% and 26% or more, respectively.** We measure the CPU load and temperature with MARLIN and compare these to those with default-DNN. Lower CPU load leaves more computational resources for other AR tasks (e.g., pose estimation, lighting estimation), and a lower CPU temperature means a more comfortable user experience when holding/wearing the AR device. Fig. 3.9 (center and right) shows that in 60% of the cases, the CPU load and temperature are reduced by at least 10% and 26%, respectively (averaged across all 8 cores of the Google Pixel 2 phone). Despite the CPU's cooling technology and operation in a temperature-controlled 20°C room, MARLIN reduces the CPU temperature by 4.88° on average (Fig. 3.9 left).

MARLIN **significantly helps in coping with CPU frequency throttling.** Automatic CPU throttling lowers the CPU frequency based on the load to help conserve energy and reduce the temperature of the chip, and is enabled by default on recent smartphones. While we did not observe CPU throttling on the Google Pixel 2 phone (due to several optimizations [65, 1]), we investigate how MARLIN performs when compared with default-DNN on older processors. Our goal is not to reduce throttling on mobile devices in general, for which methods exist (e.g., [88]), but rather to reduce throttling in the context of object detection and tracking, especially on less powerful mobile devices. Towards this, we next perform experiments on the LG G6, which has a slightly older processor (Qualcomm Snapdragon 821). On this phone, we see that all 4 CPUs work at full speed when executing the DNN, and are automatically throttled after a few minutes of execution. The CPU frequency drops from 1.6 to 1.06 GHz on the two little cores and from 2.35 to 1.06 GHz on the two big cores [74]. Because of this, the power consumption is reduced for default-DNN as shown in Fig. 3.12b, but MARLIN further improves energy efficiency on the CPU-throttled phone (more power reduction).

Interestingly, we find that CPU throttling causes a 2× increase in the DNN execution latency (taking 1221-2553 ms to execute) and a 80% increase in the object tracker's execution latency (taking 24 ms-43 ms). Thus, DNN-based detection fails more frequently because the scene has already changed by the time the result is returned, especially in moderate to fast motion videos. Figs. 3.12a and 3.12b depict the significant decrease in accuracies as compared to a non-CPU-throttled phone; specifically, default-DNN takes a hit of 49.2% in ACP and 54.0% in IOU when throttled. MARLIN triggers the DNN less of-



(a) Google Pixel 2 (b) LG G6 (c) Relative accuracy

Figure 3.12: On a phone with automatic CPU throttling, MARLIN improves accuracy compared to default-DNN.

	Method	Accuracy		Energy Consumption	
		ACP (%)	IOU (%)	Battery drop (%)	Power (mW)
Live 1	Default-DNN	90	61	11	1724.55
	MARLIN	92	61	3	319.54
Live 2	Default-DNN	80	56	11	1710.49
	MARLIN	87	51	5	880.65

Table 3.4: In live experiments, MARLIN saves significant energy with similar accuracy to default-DNN.

ten, reducing the frequency of CPU throttling, and this improves the accuracies on average. We see this when we compare the relative accuracies of default-DNN and MARLIN on the CPU-throttled phone: for 80% of the videos, MARLIN has a higher ACP and IOU, by an average of 44.0% and 38.7%, respectively (Fig. 3.12c).

3.5.3 Live Experiments

To showcase MARLIN's proof-of-concept prototype and evaluate its real-time performance, we perform live experiments in our lab. We train the object detector to detect and overlay virtual objects on people, using VOC2007, VOC2012 [36], and Penn-Fudan Pedestrian [118] datasets for training. We load the trained DNN onto two identical phones (Google Pixel 2), configuring one to run default-DNN and the other, MARLIN. One person holds the two cameras side-by-side, and we request a few student volunteers (2-3) to appear in front of the cameras and act as specified in the scripts shown in Table 3.5 and a screenshot is shown in Fig. 3.13. Each trial lasts 30 minutes and the process was approved by our institution's IRB.

Since we do not have ground truth for these live experiments, we use a more powerful DNN-based tracking by detection algorithm (YOLO [95]) to analyze the video offline on a 12-core Intel Xeon server with 32 GB of memory, and generate annotations considered as ground truth. We also visually inspect a subset of the results to confirm that this is in fact the ground truth.

In live experiments, MARLIN uses only 18% of power consumed by default-DNN with negligible loss in accuracy, running at 29-30 frames per second. 30 frames per second is considered good real-time performance for object tracking [31]. Table 3.4 compares MARLIN's performance with that of default-DNN. In both trials, MARLIN achieves comparable accuracy to that of default-DNN while significantly saving energy. Note here that when measuring the energy, we are careful to remove the consumption caused by auxiliary factors (e.g., the screen and the camera), which are common to both default-DNN and MARLIN. In the first trial, MARLIN uses only 18% of the power compared to default-DNN, and in the second trial, MARLIN uses 51% of the power. The second trial consumes more energy because the human subjects in that trial were slightly more active (more motion). Both MARLIN and default-DNN achieve comparable accuracy in terms of ACP and IOU.

Downloadable software: Our software is downloadable from the project website [10] and tested on smartphones. Both MARLIN and default-DNN methods are provided to enable a relative comparison between the two approaches. Note that when testing with much older phones, they may heat up and cause CPU throttling, impacting both schemes.

3.6 Discussions

Classification accuracy: If the DNN is not trained sufficiently and does not achieve high classification accuracy, this may result in mis-labeling of objects in the scene, and cause the object tracker to either (a) track the wrong objects, or (b) track the right objects but with the wrong label (e.g., track a sheep which is mis-labeled as a horse). Quantitatively, this will manifest itself as low average IOU, since having the correct object label is necessary for a non-zero IOU (see the IOU definition in §3.5.1). We have observed such scenarios in initial experiments (later corrected) when Tiny YOLO was not trained for a sufficient number of epochs, resulting in low classification accuracy, and causing MARLIN's object tracker to track the wrong objects. In future work, we plan to further investigate the relationship between classification accuracy and MARLIN's performance, and distinguish between cases where IOU is low due to poor classification or object localization.

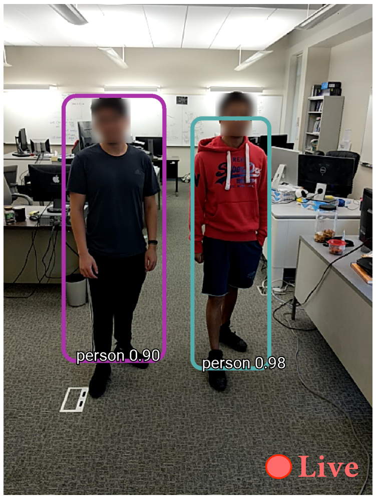


Figure 3.13: Screenshot of the MARLIN app tracking "person" in the view

Minute	Live 1	Live 2
0-5	P1 stands with minor movements	P1 sits and P2 stands
5-10	P2 enters and stands casually	P3 enters and walks in random directions
10-15	P1 and P2 walk criss-cross	P1,P2,P3 walk in and out of the camera's field of view
15-20	P1 leaves; P2 walks in random directions	Camera moves towards and away from P1,P2,P3
20-25	P2 returns; P1,P2 walk in random directions	P1,P2,P3 walk in random directions within the camera's field of view
25-30	P2 leaves; P1 walks in random directions	P1 leaves; P2,P3 walk in random directions as the camera moves

Table 3.5: Live experiment action scripts. P1, P2, P3 are volunteers.

Latency of detecting new objects in the scene: When new objects enter the scene (e.g., a person enters the room), MARLIN's change detector (Sec. 3.3.4) is responsible for detecting that change and triggering a new DNN execution. Since MARLIN uses Tiny YOLO (or other compressed DNNs) as a key component of the system, its performance cannot exceed that of the compressed DNNs in use today; in other words, it cannot detect objects that its constituent DNNs cannot, or even for detected objects, the detection latency cannot be less than that of Tiny YOLO. Qualitatively in our live experiments, we have observed this limitation with both MARLIN and with the baseline Tiny YOLO with continuous execution. However, as researchers develop new DNN models with reduced latencies, MARLIN will automatically be able to leverage these advances by swapping in new, improved DNN models into MARLIN's framework.

3.7 Related Work

Mobile deep learning: MCDNN [49] chooses which DNN to run given accuracy, latency, and energy requirements of the mobile application. Other efforts speed up DNN inference (e.g., quantized models [50], IDK cascades [120], DeepMon [55]), but only focus on detection and not the use of tracking to reduce DNN invocations. Recent works in computer vision [127, 63] combine detection and tracking, but use expensive DNN-based tracking, frequent fixed interval DNN executions, or offline knowledge of entire video clips. In contrast, MARLIN runs in real-time and adapts DNN executions based on the scene content.

Mobile AR: Liu et al. [76], Gabriel [48], and Glimpse [31] have proposed cloud/edge-based AR, among others [48, 59, 125, 124, 92]. In contrast, MARLIN focuses on energy efficiency when AR processing is run locally on the device without offloading. Further, Liu et al. [76] focus on partitioned DNN executions on an edge server, by modifying the video encoding parameters, whereas MARLIN considers local execution without video encoding. MARVEL [30] studies energy-efficient AR, assuming the location of the objects in the environment are pre-annotated, while MARLIN studies how to detect and track these objects in the first place. ARCore, ARKit, and ARToolKit [44, 15, 16] provide less sophisticated object detection for planar, non-moving objects, while Vuforia [116] can detect and track up to 20 specific instances of 3D objects. Wagner et al. [117] combine object detection and incremental tracking, but can only detect a single object in the frame.

Change detection: Using the sum of absolute differences is a naive method of change detection, and is susceptible to noise from illumination or background changes [5, 91]. Background/foreground subtraction methods using GMM [128] and KNN [129] 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. [38]); however, the feature extraction step of such methods are heavy-weight and unsuitable for mobile devices.

Hardware acceleration: There are methods that use specialized hardware sensors to either perform change detection [83] or to tune the energy usage [73]. Qualcomm and Google are developing proprietary chips for computer vision [89, 46]. Such advances are complementary to MARLIN.

3.8 Conclusions

Energy consumption is a major concern for AR. We design MARLIN, a framework to reduce the energy consumption of object detection and tracking, which are important in the AR computational pipeline. MARLIN intelligently alternates between DNN object detection and lightweight incremental tracking to achieve high accuracy while saving energy. Our Android prototype shows that MARLIN drastically reduces energy consumption (up to 73% savings) with a minor accuracy penalty (at most 7% for 75% of the test videos), and surprisingly, in 46.3% of the cases, improves both accuracy and energy compared to a default system using DNNs continuously. Future work includes incorporating inertial odometry to further reduce energy consumption.

Chapter 4

Power-efficient Infrastructure-free Collaborative Mobile Augmented Reality

4.1 Introduction

Collaborative or multi-user AR experiences are on the rise, with examples including Pokemon Go's Buddy Adventures feature [97], Google's Just a Line virtual graffiti drawing app [40], Meta-AR-App for education [115], and AURORA for battlefield scenarios [35]. These apps enable joint sessions of co-located users, who view and interact with the same set of holograms, which are fixed in place with respect to real world objects (*e.g.,* virtual graffiti fixed to a particular wall).

While multi-user AR apps typically rely on cloud/edge infrastructure to help with heavy computations and sharing of information across devices, such infrastructure may be unavailable in many cases; e.g., a search-and-rescue or an ad hoc AR game at a remote beach. In the first example, AR users may need to see maps or overlaid instructions on their real world view to navigate the terrain and in the second, users may interact with a virtual beach ball. In this chapter, we envision multi-user AR apps working natively on mobile devices even without access to edge/cloud infrastructure.[1]

A key concern in infrastructure-free AR is power expenditure, which affects experience and application longevity. Specifically AR entails two sources of high-power computation which can drain device batteries. First, the AR app has to compute where to place a hologram, and record the hologram's location with respect to the real world. Second, the AR app has to localize itself in the real world, to correctly display holograms that appear within a user's field-of-view (FoV) (*e.g.,* so as not to occlude real objects). Recent work uses deep neural networks (DNNs) [78, 76, 13] for the first task, and simultaneous localization and mapping (SLAM) is commonly used for the second task [70, 43, 94]. As shown in prior work [30, 13] and seen in our measurements, these are power hungry; e.g., SLAM consumes roughly 1.01 Watts and a DNN execution consumes 0.98 Watts on a Google Pixel 4; this is comparable or higher than video streaming [131, 114].

In this chapter, we present COLLAR, a system that enables power efficient collaborative, infrastructure-free AR apps by exploiting opportunities for energy savings by reducing redundant or similar computations performed across devices. With COLLAR, not

[1] While there are emerging efforts on infrastructure-less AR [14, 82], they are proprietary and their performance is not well understood.

every device has to continuously run all computations. Rather, heavy computations performed by a primary device are repurposed by others (secondary devices); the role of the primary is rotated to distribute the energy drain. We call this *collaborative time slicing.* COLLAR ensures that the AR experience in terms of hologram placement accuracy is similar to the case where all devices perform their own computations (thus, all expending high power). To the best of our knowledge, we are the first to design a fully decentralized, power-efficient, and infrastructure-free collaborative AR framework, unlike approaches that rely on the cloud or edge like [30, 19],

Realizing COLLAR in practice entails three key challenges:

- *Synchronization inconsistencies.* Holograms can appear at inconsistent locations in the FoVs of different devices, if the secondary devices' understanding of the real world are not synchronized with that of the primary. This can happen if the primary or secondary devices select the wrong frames for synchronization.
- *Failures due to abrupt motion.* Synchronization across devices may be lost and/or there could be changes in the appearance of an object in the FoV, if there are sudden and abrupt changes in a AR user's orientation or location (*e.g.,* a user turns right all of a sudden).
- *Cold re-starts can disrupt the experience.* We may need to fallback to heavyweight computations (e.g., during primary rotations). Cold re-starts of these computations incur high latency and can disrupt the user experience.

In designing COLLAR, we make key contributions to tackle these challenges:

- **Robust coordinate system synchronization.** We develop a method to synchronize the 3D coordinate systems of collaborating AR devices, enabling them to efficiently share hologram positioning information. Our approach considers time dynamics, where we filter stale information and suppress unreliable outliers, which markedly reduces synchronization errors and improves the hologram placement accuracy compared to state of the art methods. [2].
- **Rapid local repair to cope with abrupt motion.** To cope with disruptions due to rapid unforeseen motion, we incorporate a local repair method that stores previously seen templates of the objects that holograms are attached to, and tries to match current object views to those templates to reidentify objects lost from view. Using this method, COLLAR is 28× faster in recovering objects, compared to an approach where a DNN is triggered for the same purpose; this also yields power savings of 39.9%.
- **Faster world re-connection after failure.** Re-initializa-tions of the framework will infrequently be needed, due to either the primary device's energy drain (*i.e.,* a new primary is to be chosen) or because of a local repair failure. COLLAR incorporates a custom SLAM instantiation approach for expeditious recovery. In brief, it uses stored information from prior stable instances for speed up. This yields a 4.4× speedup compared to when the secondary has to completely re-initialize everything from scratch.

[2] We point out here that in this work, since the holograms we consider are bounding boxes outlining the object of interest, we consider the object detection accuracy (IOU) as the measure of hologram placement accuracy, similar to previous works [76, 92].

We implement a COLLAR prototype on Android. It works on multiple smartphones, without needing root access. Our code is available at an anonymous website [9]. We compare COLLAR's performance with two academic state of the art systems, MARVEL [30] which uses edge infrastructure, and MARLIN [13] where power optimizations are done for on-device computations (no edge is involved). Our evaluations in various representative scenarios show that (i) COLLAR reduces power by 29% *and* improves the object detection accuracy by 39% in terms of IOU, compared to MARLIN, and (ii) COLLAR improves the object detection accuracy by 78% in terms of IOU with a 42% increase in power, compared to MARVEL (which has the benefits of edge infrastructure).

4.2 Motivation

Example: Consider a scenario (Fig. 4.1) where AR-equipped firefighters navigate a building to search for people needing rescue. When the lead firefighter finds a person, the AR device automatically detects and highlights the person on the firefighter's display. When supporting firefighters arrive, the person is also highlighted on their displays. If the person or a firefighter moves, the highlights must update correspondingly on individual AR displays.

A naive approach in the above scenario would be for each AR device to operate independently, scanning the environment for people and tracking their locations, resulting in redundant computations across devices and wasting energy. In this chapter, we explore a simple idea: a primary AR device "takes charge" of the bulk of the monitoring and location tracking, while other secondary AR devices turn off heavyweight computations to save energy. In the example, the lead firefighter's AR device could initially act as the primary, and when the supporting firefighters arrive, their devices receive information about the person and the environment and display the highlights. Note that the secondary devices perform minimal computations – only those needed to update the display as the firefighters or person moves. Subsequently, one of the supporting firefighter's device can take over as the primary, if needed to prevent battery drain.

Single-user operation: The above example involves three main computation steps.

1. **Locate:** Locate objects of interest (e.g., people), typically done using DNNs [78, 76, 13].
2. **Track:** Keep track of the locations and orientations of an AR device, typically done using SLAM [70, 43, 94].
3. **Render:** Render the holograms on the AR device when the person is within the firefighter's field-of-view (FoV). This is done by combining information from the first two steps (where the object of interest is, where the AR user is) and using computer vision techniques (§ 4.4.1).

With the naive method, the energy consumed on each device will be the aggregate from the above three steps. In our experience and as reported by others [124], rendering is typically lightweight given the low complexity of today's 3D holograms. Thus we focus on the energy consumption of the first two steps.

Energy costs: We empirically measure the energy consumption of the naive strategy where each AR device operates independently and runs SLAM or DNNs on nearly every frames, which is commonly done [43, 70, 13, 76]. We perform measurements on several

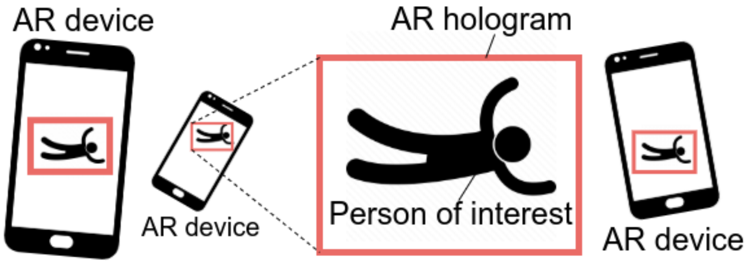


Figure 4.1: Multiple AR devices view the same person of interest and the attached AR hologram.

smartphones (Google Pixel 4/4a/5 and Samsung S21), using VINS-AR [70] as the SLAM framework and EfficientDet [107] on Tensorflow Lite as the DNN. Table 4.1 shows the energy expenditure of running SLAM alone (1.2 W), DNNs alone (1.2 W), and SLAM with DNNs and object tracking simultaneously (2.4 W). Even with object tracking to help reduce energy expenditure, running SLAM and DNNs consume high energy. Note that this is energy consumed by a single user device; with N users would consume approximately $\geq 2.4N$W of power, not including the energy for communication between users.

A case for sharing: We observe a natural opportunity for energy savings – sharing common information about the object/hologram locations. This is possible since the users engage with the same AR ecosystem (*e.g.,* same area of interest), view common objects in the real world, and interact with the same virtual holograms. Hence their heavyweight computations relating to localization in the common ecosystem and detecting the real world objects, can be shared. Returning to the firefighter example, the primary can share the person's 3D coordinates with the secondary devices when they arrive, enabling them to quickly re-identify the person and display the highlights, without running DNN or SLAM.

Operation	Power (W)	Operation	Power (W)
OS+Camera+Screen	3.016 ± 0.239	Optical Flow [20]	0.319 ± 0.072
IMU Tracking (§ 4.4.2)	0.361 ± 0.151	Image-Based Localization (§ 4.4.2)	0.994 ± 0.438
WiFi P2P Send	0.166 ± 0.033	SLAM [70]	1.208 ± 0.164
WiFi P2P Receive	0.085 ± 0.027	DNN [107]	1.225 ± 0.308
Local Repair (§ 4.4.3)	0.650 ± 0.105	SLAM+DNN+OF	2.424 ± 0.402

Table 4.1: Energy Expenses for key operations in COLLAR. Averaged measurements with Google Pixel 4, Google Pixel 4a 5G, Google Pixel 5, and Samsung S21.

4.3 System Overview

In this section, we first describe the design objectives in building COLLAR, and describe the key functions within COLLARthat address various challenges that arise in realizing its fully decentralized low power and low latency architecture.

Premise. We envision a set of users seeking to engage in a fully decentralized AR experience. As described in §4.2, the AR devices use SLAM to localize themselves in the environment, and use DNNs to detect objects in their fields of view (FoVs), thereby allowing them to place holograms at the proper locations in the environment. We expect holograms placed or modified by one device to be seen by other devices. In other words, the experience is collaborative and for this purpose, the users will all need to have a common world view, as shown in Fig. 4.1. We assume that the AR devices all have WiFi or some

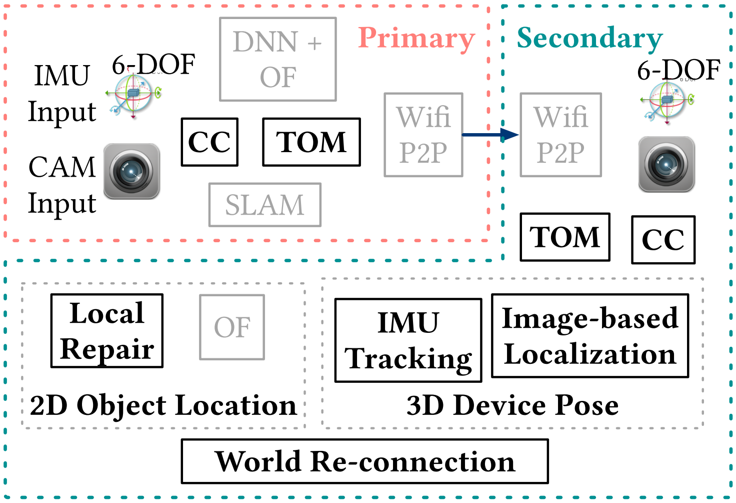


Figure 4.2: COLLAR's Architecture; CC = COLLAR Coordinator; TOM = Tracked Object Manager; OF = Optical Flow Object Tracker

other means of wireless P2P communication, and are deployed such that the users can all communicate with each other, *i.e.,* the topology is a clique. We believe that this is a reasonable assumption given the range of WiFi and the area of experience for the AR ecosystem we envision (*e.g.,* an open space such as a patio or field, or a couple of floors within a building).

In COLLAR, we use the notion of collaborative time slicing, wherein at any given time, one of the devices, which is chosen as the primary device, runs the expensive computations, while the other devices (secondary devices) leverage these computations, via the exchange of metadata with the primary device, and based on their own IMU inputs and local information. In other words, the heavy computations are collaboratively time sliced by rotating the role of the primary, to distribute the heavy energy drain. The idea of primary and secondary device, and rotating their roles is not new [3, 85].

However, the need for near real time synchronization of the users' AR worlds leads to several challenges when instantiating and managing the time slices, outlined next.

Challenges: First, in order for the primary to share 3D coordinates of the objects with the secondaries, the primary and secondaries need to have a common coordinate system to represent the 3D coordinates. However, since AR devices set the origin point of their coordinate systems to an arbitary location [43, 70], new methods are needed. This motivates COLLAR's *coordinate system synchronization* module (§4.4.1), which uses visual features to synchronize the origin points.

Second, when secondaries turn off their high-power computations, they still need to update their displays if the object or the user moves. Keeping accurate track of the device location is difficult due to accumulated IMU drift. This motivates COLLAR's *lightweight localization* module (§4.4.2), which combines IMU with camera inputs.

Third, in cases of failure due to abrupt motion or changes in appearance of the object, the secondaries should still be able to view the holograms. This motivates COLLAR's *local repair* (§4.4.3) and *world re-connection* (§4.4.4) modules, to correct small and bigger failures, respectively.

Corresponding to the above challenges, we aim to have COLLARfulfil the following properties: (Property A) tight coordinate system synchronization between the AR devices, to ensure that all users nearly always have the same views of the holograms; (Property B) low power consumption for all the secondary devices (the primary still performs the high power computations, but the role is rotated) and (Property C) quick recovery or fall back when the synchronization fails due to subtle or abrupt motion.

Timeline of operations. Fig. 4.2 illustrates COLLAR's architecture, and Fig. 4.3 shows a timeline of its operation. Below, we provide a high level overview of COLLAR's operations before we delve into the details of its components.

Initialization (t=0). Initially, both primary and secondary devices run the expensive SLAM and DNN, continuously. Each secondary device performs *coordinate system synchronization* (CSS) with the primary, after which the primary sends information about each object of interest. If the synchronization is tight, each secondary identifies which of the objects within its FoV match those advertised by the primary, and display the associated holograms, achieving Property A. If the matching is a success for all the objects of relevance, the system is considered to have reached a steady state.

Secondary devices transition to low power modes (t=1). At this point, to save power and achieve Property B, the secondary devices turn off SLAM and DNN and rely on their own camera and IMU outputs to localize themselves and other objects in the AR ecosystem as they move around, without relying on the primary. This process is referred to as *lightweight localization.* The primary continues to execute SLAM and DNN; it shares updated tracked object information with the secondary devices as needed.

Dealing with disruptions (t=3). If a secondary user (*e.g.,* Secondary B in Fig. 4.3) moves abruptly or the object itself moves, lightweight localization may fail and the hologram may be lost. In such cases, a secondary triggers a *local repair* procedure to recover the object's location and re-draw the associated hologram, towards Property C of failure recovery.

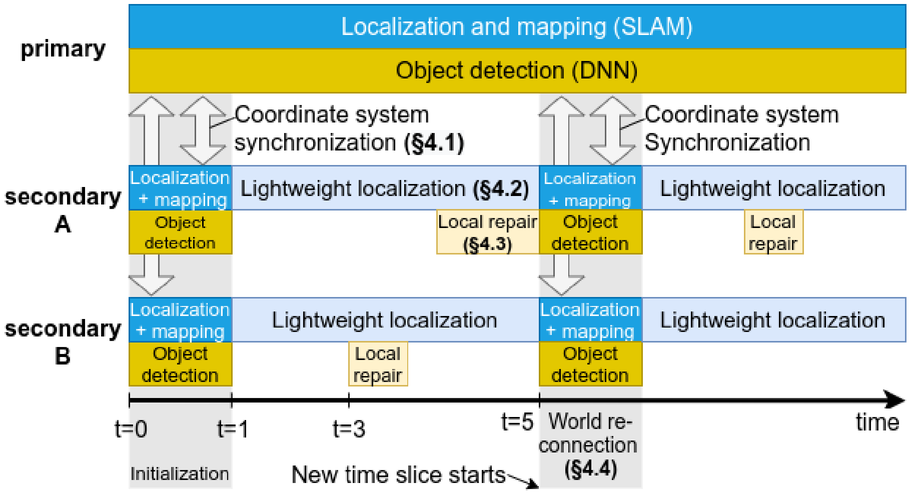


Figure 4.3: Timeline of COLLAR's operations. The primary runs heavyweight localization and object detection continuously, while the secondaries run lightweight localization and local repair in between the heavyweight computations.

Fallback and primary rotation (t=5). Finally, if the primary's residual energy drops below a threshold, or if unacceptable tracking accuracy is experienced, the primary device initiates a new time slice and returns to the initialization phase. In this case, each secondary device will need to re-initialize SLAM. Instead of a cold start, secondary devices utilize information from prior initialization phases to expedite such a re-initialization significantly (towards Property C).

Coordinator: The above phases are orchestrated by a COLLAR Coordinator (CC) running on each device. They are responsible for assigning and enforcing the primary and secondary roles, handling messaging, and triggering local repair, world re-connection, and role re-assignment. We currently assign the device with the most residual energy is chosen as the primary (ties are broken randomly); we defer advanced strategies [25, 29] to future work.

4.4 Detailed Design

In this section, we provide a more in-depth view of the COLLARcomponents described earlier.

4.4.1 Robust Coordinate System Synchronization

In COLLAR, the primary must be able to describe the location and orientation (*i.e.,* pose) of an object (and its associated hologram) to a secondary, so that the latter can easily locate the same object in space, and draw the appropriate holograms. Towards this, the primary and secondary need a common coordinate system to represent the object's pose. However, the origin of a device's coordinate systems, as established by SLAM, is typically set to an arbitrary location (*e.g.,* where the device first opened the AR app). For a unified AR experience, these coordinate systems must be tied together, *i.e.,* they need to be synchronized.

Upon experimenting with existing synchronization methods (specifically, the Perspective-n-Point method of SLAM [70]), we found that these methods often returned poor synchronization results, resulting in mis-aligned coordinate systems and the 3D object coordinates being interpreted differently among devices. The root cause of this problem was that the AR users were never perfectly still; so, the devices were constantly collecting new camera frames of the environment, and selecting unsuitable frames for synchronization.

Key idea: COLLAR builds on existing synchronization methods, but considers the *temporal aspect* when selecting frames for synchronization. Specifically, it chooses frames from the primary and secondary that are (a) similar in appearance, and are (b) close in time. If these two frames are similar in space and time, then the synchronization method can conclude that they were taken from similar viewpoints, and hence, align the coordinate systems. We call the selection of suitable input frames as *staleness avoidance*. COLLAR also checks the quality of the synchronization method's output, by checking how well the camera frames fit with the proposed synchronization. We call this *variance suppression*. Below, we describe these two components in more detail.

Staleness Avoidance Recall that we are looking for frames from the primary and secondary that are similar in appearance and time, to use as good inputs for synchronization. Since the coordinate system synchronization is fairly heavyweight, it takes place on the primary. Thus, we have access to the entire history of the primary's frames, but only the most recent frame from a secondary; hence we need to search for the best match from the primary's history. To do this, we compute two scores:

- An appearance similarity score, $s[i]$, where i is the index associated with the primary's candidate frame. The similarity is based on the number of matched visual BRIEF features between primary's frame i and the secondary's most recent frame [37].
- A time score that downweighs old frames, $0.99^{(t-t_i)/s}$, where t is the current time, t_i is the timestamp associated with frame i, and s is a normalization factor.

Combining the appearance and time scores together, we select the primary's frame i^* that satisfies:

$$i^* = \text{argmax}_i s[i] * 0.99^{(t-t_i)/s} \tag{4.1}$$

To summarize, we select a frame from the primary to maximize a multiplicative combination of appearance similarity and time freshness, compared to the secondary's frame. The two frames are processed before being input to the Perspective-n-Point (PnP) synchronization method.

PnP synchronization method PnP is a well-known technique [106], so we only briefly summarize its inputs and outputs here. First, we pre-process the frames chosen by staleness avoidance, selecting the intersection of visual features from those two frames. PnP uses this information to compute a 4x4 homogeneous transformation matrix $R_{p \to s}$, which is used to transform the 3D coordinates of an object from the primary's coordinate system p to the secondary's coordinate system s, as follows:

$$x_s = R_{p \to s} x_p \tag{4.2}$$

where x_p, x_s is a vector of 3D coordinates of the object in the primary and secondary's coordinate systems, respectively.

Variance Suppression This step checks the output of the synchronization method to see if the results are good. Since we do not have ground truth knowledge of the synchronization quality in real time, we approximate synchronization quality by its consistency. Essentially, the PnP synchronization method fits a linear model ($R_{p \to s}$) to visual features from the camera frame inputs. If more of the visual features fit the linear model, this means the linear model is consistently able to explain the observed data, and thus the synchronization is more likely to be accurate. To use this information, COLLAR counts the number of visual features that fit the linear model (within a tolerance threshold), which are called inliers [106].

It only accepts new coordinate system synchronization outputs if the number of inliers is greater than the previous synchronization attempts. Once synchronization is complete, the secondary device runs object tracking to keep track of the object found at (or near) x_s.

4.4.2 Decentralized Low-Power Localization

Once coordinate system synchronization finishes, the secondary devices turn off SLAM to save power. However, they need to continue to localize themselves in the physical world, in order to decipher any updates from the primary about object coordinates. A common lightweight localization method is using IMU sensors, which consumes markedly lower power than using full SLAM (4 × lower as shown in Table 4.1). However, IMU-based localization can acculumulate drift over time and become inaccurate over a period of time (10-20 s) [68]. This causes an unwanted disconnect between the device location within the AR ecosystem and the real world, and hence, a deviation in the position of the holograms shared by the primary (away from the actual physical objects they should be attached to).

Key idea: Our main idea is to augment IMU-based localization with visual information from the camera, and with localization information previously collected by SLAM during the initialization phase. Combining all these sources of information enables COLLAR to recover high localization precision. In addition, we also incorporate the magnetometer and gravity estimation as external references to help correct basic IMU-based tracking. Taken together, these hybrid methods consume low power and are able to provide localization results with extremely low latency but high accuracy as the AR user moves. We next describe these components of COLLAR in more detail.

Augmenting IMU Tracking with Visual Information Our experiments suggest that orientation tracking is typically stable and accurate, but the translational tracking can accumulate errors over time. This is due to translation estimation requiring a numerical double integration over the accelerometer readings, while rotation estimation only require a single numerical integration over the gyroscope readings [68]. To overcome this, if the cumulative IMU-based translation (z) exceeds a threshold, the secondary device captures the latest camera frame and uses it to re-localize within its own coordinate system. Setting this translation threshold too high can cause a large localization drift; setting it too low will cause frequent re-localizations and high energy (see Table 4.1). We empirically set the threshold to 20cm.

Elaborating on the re-localization procedure, the approach is similar to our coordinate system synchronization method above, except that instead of comparing the secondary's most recent keyframe with the *primary*'s entire history of keyframes (Eqn. 4.1), we compare the secondary's most recent keyframe with the *secondary*'s entire history of keyframes, and use that as the input to the PnP synchronization. After re-localization runs, z is reset to zero.

Augmenting IMU Tracking with Gravity Estimates We integrated several publicly available IMU-based tracking methods [7, 62] with our test AR app, but found they were unable to perform well. Specifically, when we placed a holographic cube in the world and moved the device to the left or the right, the cube either followed the screen or drifted away to another direction (when it should have stayed fixed in the physical world). Therefore, we were motivated to develop a custom IMU-based tracker as part of COLLAR.

Our design is inspired by two prior efforts: Shen et al. [103], providing a method for gravity removal from accelerometers under an arm motion model, and Solin et al. [104], tracking devices under legged/wheeled motion models. Neither directly applies to our tracking requirements – we seek to track the device in 3D, with the the user holding the device potentially moving her arm or walking around. We can borrow insights from these works and combining their ideas as follows:

1. When the device is mostly motionless (acceleration $(a_x, a_y, a_z) < 0.2m/s^2$), we estimate the gravitational forces in the three dimensions viz., (g_x, g_y, g_z).
2. When the user starts moving the device, we estimate the linear accelerations (la) as $la_x = a_x - g_x, la_y = a_y - g_y,$ and $la_z = a_z - g_z$
3. Using the standard physics kinematic equations [68], we estimate the translation every $\Delta t = 10$ ms.

Our results show that this simple method works better than other open APIs such as Android API [7] or FSensor [62], under the assumption that the user moves and stops the device occassionally (which provides a chance to re-estimate gravity and set zero velocity).

We also develop a custom orientation tracker based on [86], which augments the gyroscope with magnetometer inputs, which are known to be more stable than the gyroscope [122]. Specifically, our orientation tracker numerically integrates the gyroscope sensor readings to produce orientation estimates over time, relative to an original reference orientation. To prevent large rotational tracking drifts, COLLAR periodically checks the consistency between the magnetometer and gyroscope readings. If inconsistency is detected, it resets the gyroscope reference orientation using the current magnetometer reading.

4.4.3 Low-Latency Local Repair

Sometimes when the user motion is abrupt, a secondary device can lose track of a locally tracked object; for example, a quick turn can change the appearance of an object such as a chair. A naive way to recover from such a loss is to trigger a DNN for object detection or run SLAM; however, this consumes high power and more importantly induces long delays for the system to return to a steady state.

Key idea: Our idea here is to collect templates of the object of interest, to which the AR holograms are attached, during the initialization phase, and use those templates later on during the steady state phase to quickly re-locate objects and re-draw the holograms, without needing to re-execute DNNs. These templates are unique to each secondary device, as they represent what the object looked like from the secondary's own FoV, making local repair more likely to succeed. For example, if the primary is very close to the object of interest, it will appear very large in the primary's FoV; but if a secondary is further away from the object, it should only search for a small version of the object in its FoV. We next elaborate on the two components of COLLAR's local repair that accomplish this, viz., intelligent template collection, and fast template matching, which take $<$ 600 ms to complete.

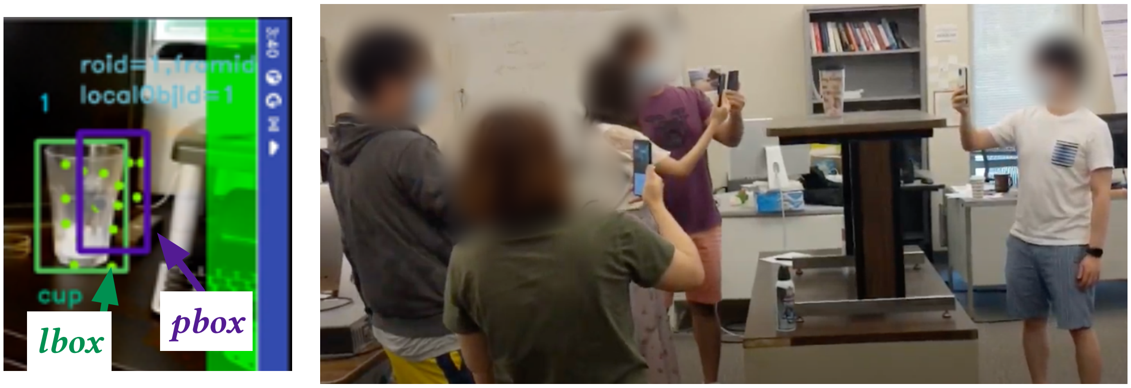
Intelligent template collection: When the DNN is running and accelerometers suggest that the device not in motion (*e.g.,* during zero velocity periods mentioned in §4.4.2), a secondary device obtains a candidate object template from the current camera frame. It uses a color moment hash [108], which is a compressed representation of the image that is quick to compute (10ms). COLLAR needs to collect a diverse, yet compact set of templates to represent a given object. It uses either of the following two criteria to accept a new template: (1) the minimum hash distance compared to all the previous templates must be greater than a threshold or, (2) the template's width and height, are greater than those of all the previous templates by a threshold. This allows the local repair to store representative templates with various perceptual features and dimensions.

Fast template matching: When a secondary loses an object it is tracking (we call this an *lbox*), and the primary sends an update about the same object (we call this a *pbox*, identified by a common ID), an object search is triggered. COLLAR first retrieves the templates associated with that object, then waits for the device to be relatively motionless, and opportunistically executes template matching; this ensures that the captured camera frame is likely to be sharp (no motion) and therefore appropriate for matching. The template matching is performed by applying a sliding window of the template's size to the camera frame, and finding the location where the sum of square differences (SSD) is minimized. This computation is repeated for each template, and the template with the lowest SSD is selected. Returning to our previous example, ideally, the smallest template should have the best match, and hence the device will draw a smaller hologram accordingly. We experimentally find that template matching takes $\approx$ 60ms, and the object location is determined with high precision (*e.g., IOU* ≈ 0.7).

4.4.4 Seamless World Re-Connection

In some rare cases of failure, local repair may not succeed, and COLLARwill start a new time slice and fallback to the initialization phase, *i.e.,* all the devices will need to execute SLAM and DNN again. A key challenge is that re-initializing SLAM naively can either cause it to reset, or fail to reconnect with its previous state and crash. Resetting SLAM from a blank state clearly misses on opportunities to leverage previously stored data, and incurs high latency. Naively attempting to merge with SLAM's previous state, for example by inputting the most recent sensor data, usually fails because SLAM expects a continuous stream of data from the camera frames and the IMU inputs; because the secondary device has not been running SLAM during lightweight localization, the data timestamps upon re-initialization have a large temporal discontinuity, causing SLAM confusion and failure. In short, a relatively long disruption occurs when naively attempting to re-initialize SLAM.

Key idea: COLLAR leverages an existing technique inside SLAM, called loop closure [70], to "trick" SLAM into merging the information from the current and previous initialization phases. Loop closure is normally used to determine when a user re-visits a previously seen area (*e.g.,* by walking in a loop). By setting the appropriate parameters, we give SLAM the impression that the device was simply lost during the lightweight localization period, and is now re-visiting the area from the previous initialization phase. Loop closure helps stitch these two worlds (from the current and previous initialization phases) together automatically, allowing for their smooth reconnection. This speeds up the current re-initialization, leading the COLLARto a new low-power steady state with significantly shorter latency than naive approaches. In the interest of brevity, the implementation details are omitted here.



(a)

(b)

Figure 4.4: (a) Screenshot of a low-power secondary device; the green rectangle is the locally detected bounding box (lbox) and the purple rectangle shown (used for debugging), is the matched bounding box from the primary (pbox). (b) 5 users engaging with the COLLAR app in our experimental setup.

4.5 Implementation

In this section we describe the implementation of COLLAR.

Platforms: We implement COLLAR on smartphones running Android 11 (Google Pixel 4, Google Pixel 4a 5G, Google Pixel 5, and Samsung S21). We use VINS-Mobile [70], TensorFlow [109], and OpenCV [23] libraries to implement SLAM, object detection and tracking, and PnP synchronization and image processing, respectively.

Module implementation: There are two main parts of COLLARviz., the main UI in Android called MainActivity, implemented in Java, and the key SLAM class called ViewController, implemented in C++. We interface these two main classes through the Java Native Interface (JNI). The *coordinate system synchronization* works inside ViewController to retrieve and match keyframes, and uses the OpenCV library for PnP synchronization and other related visual processing tasks. *Lightweight localization*'s IMU tracking is implemented inside MainActivity to access and process IMU sensor inputs. *Local repair and world re-connection* are implemented inside ViewController using OpenCV's template matching and SLAM's loop closure, respectively.

Inter-device information sharing: We use Android's WiFi P2P [8] to implement inter-device communications. We use Java's client-server TCP socket communications for unicast, and UDP sockets for broadcast. The Boost library [22] packs complex data structures (like keyframe information and object tracking information) and compress each into a file saved to permanent storage.

Logging: To estimate the power consumption, we read EXTRA_VOLTAGE and BATTERY_PROPERTY_CURRENT_NOW variables of Andorid's BatteryManager to obtain the battery's voltage (mV) and current (μA), respectively. We then calculate $Power = Voltage * Current$ in Watts. This does not need root privileges. To compute the hologram placement accuracy in terms of IOU, we log for each tracked object the following: for each pbox (1) timestamp, (2) object id, (3) the object's bounding box, (4) the object's class, and if there is a matched lbox, we add (5) matched lbox id, and (6) matched lbox's bounding box. We also save raw camera frames into permanent storage with synchronized timestamp information with the saved bounding box above. At the end of each trial, we transfer the saved log files to a server for offline post-processing, and when the app starts again, it purges all the previously stored files before the main process is called.

Baselines: We implement the following baselines:

MARVEL [30] is a single-user mobile AR system that utilizes edge infrastructure. Unlike COLLAR, it requires specialized hardware (phones with depth camera and/or LiDAR sensors) to generate an offline map and perform localization. We use MARVEL as a representative baseline that uses the edge. Because the MARVEL source code is not open sourced and we were unable to obtain the same via other means, we implement what we believe is a faithful reproduction of MARVEL, as follows. We survey the area using SLAM [70] to build an offline map of the surroundings and objects of interest. Then, the lightweight AR client app continuously localizes itself online in this pre-built map on the edge server (unlike COLLARwhich is infrastructure-free), using our implementation of Eqs. 6-8 from [30]. We call this *Centralized Localization*.

MARLIN [13] is a single-user mobile AR system that focuses on object detection. It selectively triggers DNN object detection on mobile devices if there is a significant change in the scene; otherwise, it uses an optical flow object tracker to update the object locations, with the overall goal of saving energy. We obtained MARLIN's source code and trained ML models. However, given the rapid evolution of machine learning reseearch, we replaced MARLIN's ML models (*e.g.,* Tiny YOLO) with the newer EfficientDet model [107] trained on the COCO 2017 dataset [75], which has better accuracy. COLLAR also uses this DNN model for fair comparison. We modify MARLIN slightly to achieve collaborative AR among multiple devices as follows. The primary device runs MARLIN and shares the 2D object locations (rather than 3D coordinates as in COLLAR) with the secondary devices, which also run MARLIN. Upon reception, the secondaries only display a holograms (bounding boxes) if a locally detected object matches with the shared object from the primary, using a matching test (IOU¿0.3 similar to [13]).

Vanilla runs SLAM and DNN continuously on all the devices and perform coordinate systems synchronization (as with COLLAR's initialization phase) to achieve collaborative AR without considering energy issues.

4.6 Evaluation

In this section, we present our evaluations of COLLAR. All experiments were conducted with IRB approval. We first list our metrics of interest and then present our results.

4.6.1 Evaluation Metrics

Power consumption: We log the power consumption on each phone, when it is running only the Android OS, the camera, and screen display (brightness set to 70% for all); this is referred as the base power, $power_{base}$. When evaluating COLLAR or the baselines, we run the application and log the total power consumption, p_{total}. We estimate $power_{app} = power_{total} - power_{base}$ for comparing the algorithms. This essentially isolates the power consumption from the baseline or from COLLAR, as the case may be.

IOU accuracy: The Intersection Over Union (IOU) [27] is a number between 0 and 1 that captures whether the hologram (bounding box) seen by a user is where it should be (fixed to an object). The IOU is defined as $\frac{O \cap G}{O \cup G}$, where G is the ground truth bounding box and O is the bounding box displayed by COLLAR. The larger the IOU, the better. We report the average IOU over all the analyzed frames. To obtain the ground truth bounding boxes and object classes, we execute the largest EfficientDet DNN model, EfficientDet-7x.

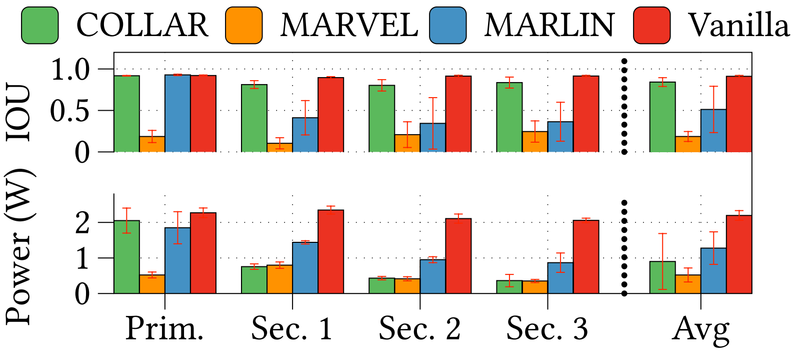


Figure 4.5: Comparing COLLAR with three baselines, where four semi-stationary users track two objects from similar FoVs. On average, COLLAR improves 78% IOU over MARVEL but with a 42% increase in power (because of the absence of edge infrastructure); compared to MARLIN, COLLAR both reduces 29% power and improves 39% IOU.

The IOU is non-zero only if the ground truth object class matches the class output by the primary/secondary's DNN. This measures whether the secondary is indeed highlighting the correct object.

4.6.2 End-to-end evaluations of COLLAR

We first provide our holistic evaluations of COLLARin various scenarios and compare its performance in terms of power consumption and IOU accuracy, with the baselines.

A semi-stationary scenario.

In this experiment, we have four volunteers holding four devices (the primary is a Samsung S21, Secondary 1 is a Google Pixel 5, and Secondary 2 and 3 are Google Pixel 4 devices). The volunteers walk around an area of $20m \times 20m$ and after initialization, point their devices with similar FoVs, to track two objects (bottle and cup). At steady state, they move the devices around 5-10cm and/or rotate them by 5-10 degrees, keeping the objects in the FoV. They remain in this steady state for 2 minutes. This reflects AR use cases where the users are semi-stationary to interact with the holograms associated to the physical objects in their FoVs (*i.e.,* bounding boxes for the bottle and cup). These experiments are repeated 5 times, first with COLLAR and then with the three baselines from § 4.5. In Fig. 4.5, we show the power consumption and IOU accuracy seen by each of the users and with all the algorithms.

COLLAR **improves the IOU by 78% over MARVEL.** On the secondary devices, COLLAR consumes power comparable to MARVEL because with both methods, these devices operate in the low-power mode. On these secondary devices, COLLAR performs IMU tracking and object tracking almost continously, while MARVEL does similar IMU tracking but also offloads images using WiFi-P2P to the server. COLLAR achieves significantly better IOU because it utilizes the object locations found by the DNN during initialization, rather than relying on MARVEL's offline stored locations. Furthermore, in most cases of failure, COLLAR uses local repair to recover object locations efficiently. In contrast, MARVEL only tries to place objects based on coordinate system synchronization, which can suffer from high variance as discussed earlier.

The total power, however, is higher with COLLAR compared to MARVEL, since in MARVEL the edge does the heavy computation whose energy is not counted (in our implementation of MARVEL, both the primary and secondaries offloads the computation).

The increase is 42% (because the primary in COLLAR runs heavy computations continuously), but we again point out that (a) COLLAR eliminates the need for the edge server and (b) we only have four devices in our experiments. If the number of devices leveraging the primary's computation are higher, this power is amortized and the penalty will be much smaller.

Compared to MARLIN, COLLAR **both reduces the power (by 29%) and improves IOU (by 39%).** Because MARLIN does not consider the 3D positions of the devices and objects, there are frequent mismatches between its 2D pboxes and lboxes; for example, this can occur when the users view the objects from slighly different distances, or slightly rotate or move their devices. Moreoever, although MARLIN can save power by not running SLAM, it often triggers DNN executions, causing higher power consumption compared to what is consumed by COLLAR's secondary devices, which run neither DNN nor SLAM in steady state.

Finally, compared to Vanilla, COLLAR has a marginally lower IOU (by 8%), because in Vanilla, all devices use heavy computations – but importantly, COLLAR reduces the power consumption by 59%. One observation we point to is that the DNN model (*EfficientDet-Lite2*) that runs on the mobile devices can achieve very high IOUs, compared to that with the heavyweight DNN model running on the server (*EfficientDet-7x*); in other words, the lightweight mobile model offers similar accuracy to its heavyweight counterpart.

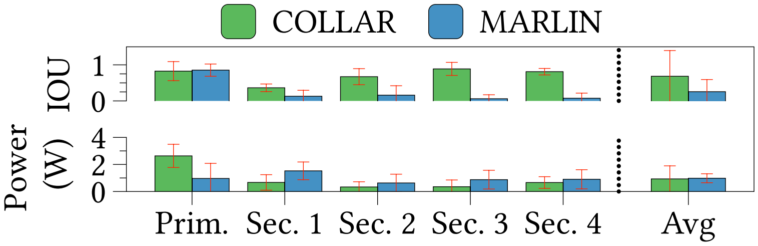
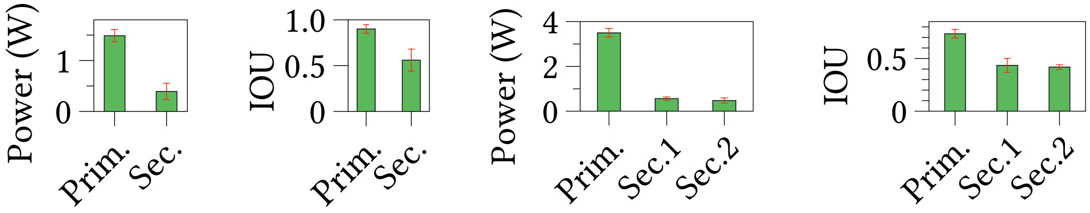


Figure 4.6: When 5 users track an object from different FoVs, COLLAR outperforms MARLIN with 64% better IOU, while consuming comparable power.

Scenarios where users have different FoVs

In this experiment, we add a fifth volunteer with a Google Pixel 4a 5G (as Secondary 4) to the previous setup; after initialization, the users stand around a table and track a single object in the middle from 5 different FoVs with semi-stationary motion. We run 1 trial and compute the average power and IOU for the duration of the experiment (with samples at the granularity of each frame). We focus on comparing COLLAR with MARLIN because the latter shows acceptable performance in terms of power and IOU from the previous section, and both are infrastructure-free.

Compared to MARLIN, when users have different FoVs, COLLARimproves the IOU by 64%. Fig. 4.6 shows that MARLIN takes a significant hit due to marked mismatches between the pboxes and lboxes arising due to the different FoVs of the users; we observe that the users are disconnected from the collaborative AR experience for significant times (as shown by their low IOUs). In contrast, COLLAR tracks the object with high IOU, because of (a) its efficient initialization that connects the 3D coordinate systems of the 5 devices together and (b) its ability to leverage IMU tracking and object templates to quickly adapt to a user's motions.



(a) Power (b) IOU (c) Power (d) IOU

Figure 4.7: (a-b) In COLLAR, a mobile secondary device can achieve good IOU accuracy. (c-d) COLLAR allows three users constantly walking around a table to collaboratively track an object with high IOU accuracy.

From the results we note that a lower IOU is experienced by COLLAR user Secondary 1. A deep log analysis reveals that this user suddenly moves the device during the initialization phase, causing the object ID to be changed; thus, the local repair does not create the proper object templates and later cannot execute successfully to update the bounding box based on the object's visual features. However, Secondary 1 is still able to use the 3D coordinates from the primary to see a bounding box (the pbox), because it has lightweight localization running, and achieves an IOU of 0.36 on average. This is lower than the other users who were able to run local repair and who achieve IOUs of 0.7-0.8; however, more importantly, collaboration helps achieve a higher IOU, even for this user, compared to MARLIN.

Secondary Device in Constant Motion

In this experiment, we run COLLAR with one primary and one secondary device, who track two objects in their FoVs. The primary user is semi-stationary but the secondary moves or rotates the device back and forth (*i.e.,* until the left object nears the left screen edge or right object reaches the right screen edge; at this point the user immediately moves/rotates the device in the opposite direction). We run 3 experimental trials, each lasting 2 minutes. In Figs. 4.7a, 4.7b, we show that in this challenging scenario of a constantly moving secondary device, the IOU drops to 0.56 (compared to 0.7-0.8 in the previous semi-stable experiments) which is still considered to be very good for object tracking [72]. Because of the motion, stable object templates can rarely be collected, and local repair is mostly unsuccessful. However, because the secondary user still has the 3D coordinates of the object obtained from the primary, it can see the objects' pboxes, and achieve good IOU accuracy.

Collaborative AR with users walking in circle

In this experiment, three users (one primary and two secondary) slowly (1m/s) and continuously (walk 1m and momentarily stop and then continue) walk around a table while always keeping one object in the FoV. One may consider this to be one of the most challenging scenarios for COLLAR because (a) object tracking can easily fail because the object appearance changes quickly due to rotated views, causing the local repair to be triggered often and (b) drift accumulates in the IMU-based translational tracker. In spite of this, Fig. 4.7c,4.7d demonstrate that COLLAR's performance is still reasonably good, as the secondaries achieve IOUs of around 0.4 (which is considered satisfactory [72]) with a power consumption of approximately 0.4 W. Compared to MARVEL and MARLIN in the much simpler semi-stationary scenario from § 4.6.2, in this more complex scenario, (i) COLLARachieves a better average IOU of 56% compared to MARVEL, and (ii) consumes 52% less power and 12% higher IOU with respect to MARLIN.

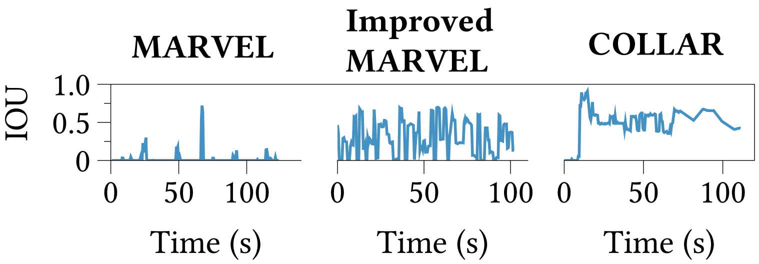


Figure 4.8: Evaluating coordinate system synchronization. COLLAR outperforms the baselines by employing Staleness Avoidance and Variance Suppression.

4.6.3 Component-wise Benchmarks

In this subsection, we evaluate the four components of COLLARfrom Section 4.4 individually, supplementing the end-to-end experiments shown thus far.

COLLAR's coordinate system synchronization improves IOU by 40% compared to MARVEL. In this experiment, we use two smartphones (Google Pixel 4 and Samsung S21) which establish SLAM and synchronize their coordinate systems and track one object in their FoVs. Fig. 4.8 shows a timeline of the secondary's IOUs (the secondary does not run DNN in this experiment), and we see that MARVEL exhibits very low IOU accuracy. This is because MARVEL does not consider the time (freshness) of the keyframes and ends up repeatedly picking stale keyframes from the primary's history, based only on visual feature similarity, leading to poor coordinate system synchronization. Specifically, a closer examination revealed that the stale keyframes chosen by MARVEL often resulted in outlier data points after the coordinate system synchronization. This motivated the development of our Staleness Avoidance technique.

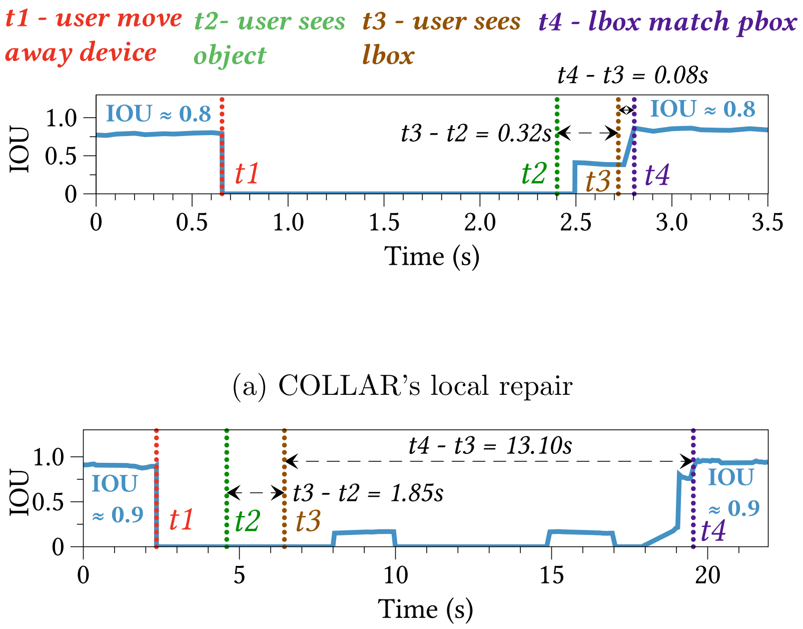
To show how Staleness Avoidance improves coordinate system synchronization, we modify MARVEL by incorporating the exact staleness avoidance technique used in COLLAR;

we call this Improved MARVEL. We find that the IOU accuracy improves significantly in Improved MARVEL, because it chooses fresher keyframes for synchronization and very few outliers after performing PnP coordinate synchronization.

However, without Variance Suppression, Improved MARVEL still does not handle high synchronization variance between different synchronization attempts, leading to a high fluctuation in terms of IOU. In other words, Improved MARVEL finds a good synchronization result initially, but gives it up too quickly and replaces it with a worse result on subsequent synchronization attempts. In contrast, COLLAR(on the right side of Fig. 4.8) quickly achieves a high level of IOU accuracy by using Staleness Avoidance, and retains this high level for a long period, because it also applies Variance Suppression. Thus, COLLAR, in the steady state, improves IOU accuracy by 40% on average, over Improved MARVEL.

COLLAR's lightweight localization copes better with user motion and achieves up to 81% better IOU, compared to centralized localization (MARVEL). In this multi-client (1-3 clients) experiment, we have one primary device and multiple secondary devices that move slowly from left to right $\approx 5cm$, stop, and then right to left, and keep going for 2 minutes. This is a use case in AR, where the users stop to interact with virtual objects and move to change FoV and interact with them again.

At steady state, with COLLAR, the secondary devices perform lightweight localization using using IMU inputs and local visual information. As the baseline, we use MARVEL's representative architecture for centralized localization: all users offload visual data to the edge computing server, to localize themselves in the prebuilt offline map. We measure the latency from when the localization request is made to when the result is received, for



(a) COLLAR's local repair

(b) DNN as local repair

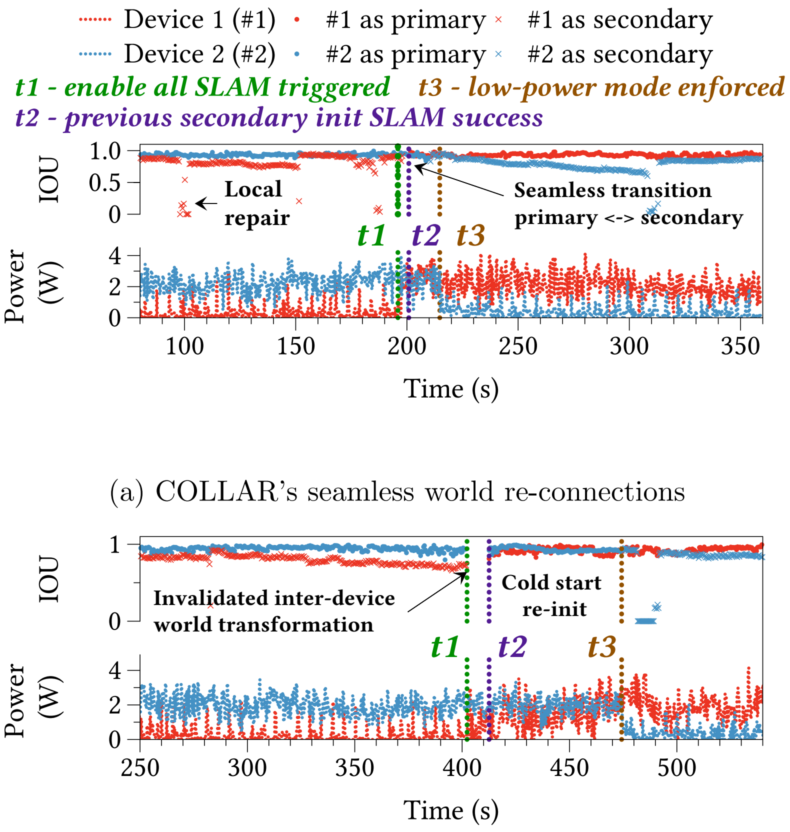
Figure 4.9: Local repair quickly recovers the hologram (bounding box) after an object reappears in user's FoV, 28× faster than a DNN.

both methods. We find that COLLAR experiences $\approx 0.22s$ of latency on average per device, no matter whether there are 1, 2, or 3 clients. On the other hand, MARLIN has average latency of $0.48s$, $0.55s$, and $0.59s$ as the number of clients increases from one to three clients, respectively. This is because MARVEL sends camera frames to the edge server over a WiFi P2P link, which can become a bottleneck due to congestion, whereas COLLARperforms lightweight localization locally in parallel. In terms of accuracy, for a two-client experiment we see that COLLAR's low latency improves the IOU accuracy by 36% (from 0.17 to 0.27) and by 66% (from 0.14 to 0.41), for Secondary 1 and 2 respectively, compared to centralized localization.

COLLAR**'s local repair recovers lost tracked objects 28 × faster than using DNNs directly for local repair.** In this experiment, we have a pair of primary and secondary devices tracking one object in their FoVs. At steady state, the secondary device suddenly fully changes its FoV (*e.g.,* turns away) and then returns to the original FoV. We run 10 trials for each method. We seek to measure the time from when the object first re-appears in the FoV, to when its bounding box appears on the screen, *i.e.,* how long local repair takes. Fig. 4.9 depicts the time evolution of IOU accuracy and other key events. The time we seek to measure is $t4 - t2$.

From Fig. 4.9, we find that COLLAR's local repair spends (A) 0.32s waiting for the device to become quasi-stationary ($t_3 - t_2$ in Fig. 4.9a) , a portion of which (0.06s) is spent on performing object template matching, and (B) 0.08 s on registering the object between the primary and secondary's versions (lbox and pbox) ($t_4 - t_3$ in Fig. 4.9a). In contrast, a baseline of triggering DNNs constantly to search for the re-appearance of the object takes $1.85s$ ($t_3 - t_2$ in Fig. 4.9b), encompassing multiple iterations of DNN execution until the object is found. Subsequently, because DNN only understands 2D object coordinates, it has to wait until the object's 2D position in the secondary's display is similar to the 2D coordinates received from the primary, in order to be confident that it is highlighting the same object. These two processes take about $14.95s$ in total ($t_4 - t_2$ in Fig. 4.9a) in this trial.

When we average these results over 10 trials, we find that COLLAR achieves local repair 28× faster than DNN. Overall, COLLAR's local repair can recover the object location (at t_4) precisely. Despite DNN's longer latency as one might expect, it offers object detection



(a) COLLAR's seamless world re-connections

(b) Cold-start world re-connections

Figure 4.10: Seamless world re-connections help quickly transition to the next time slice, 4.4 × faster than using a cold-start re-initialization.

with higher IOU both before (≈ 0.9 before t_1) and after repair (≈ 0.9 after t_4) than COLLAR's local repair which runs object template matching (≈ 0.8 before t_1 and after t_4) (Fig. 4.9). Our measurements also show that COLLAR's local repair consumes less power than DNN, at 0.84 W vs 1.41 W, respectively, on average, which is negligible.

COLLAR's seamless world re-connection enables transition to a new low-power steady state $4.4\times$ **faster than with a cold-start.** To show this, we again consider two devices and show the time evolution of what happens during world re-connection

Fig. 4.10. We would like this transition to be fast. In this example, Device 2, which is the primary device, initiates a new time slice because of a significant energy drop. With COLLAR's seamless world re-connection, the secondary device (Device 1) re-instantiates SLAM and quickly succeeds in coordinate system synchronization with the current primary device, leveraging previously stored data from the previous synchronization instance. Subsequently, Device 1 is promoted to be the new primary in the new time slice, and transits to the next low-power mode. Fig. 4.10a shows a timeline of COLLAR's world re-connection. The total time from when the secondary device re-instantiates SLAM (t_2) to until the low-power transition of Device 1 (t_3) is 13.9s. In contrast, with a cold start of SLAM as shown in Fig. 4.10b, the process ($t_3 - t_2$) takes 61.7s, *i.e.,* COLLAR's world re-connection process is 4.4× faster.

4.7 Related Work

Single-user AR: Several works study cloud or edge-based AR for a single user, including Liu et al. [76], Gabriel [48], and Glimpse [31], among others [48, 59, 124, 92]. These mainly focus on hologram placement using DNNs or other computer vision methods, without considering how the user and hologram locations relate in order to more accurately position the holograms, or deeply studying the energy consumption. MARVEL [30] and MARLIN [13] do focus on energy consumption of mobile AR. MARLIN reduces the energy of finding the right locations for the holograms, while MARVEL assumes those locations are given and optimizes the energy of a user walking around and revisiting those holograms. However, both of these works focus on energy efficiency for a single-user, rather than the

System / Features	Collab-AR [78]	Liu et al. [76]	Edge-SLAM [19]	MARLIN [13]	MARVEL [30]	COLLAR
Energy efficient				✓	✓	✓
No edge infrastructure				✓		✓
Collaborative AR	✓					✓
Real-time updates	✓	✓	✓	✓	✓	✓
Uses SLAM			✓		✓	✓
Uses DNN	✓	✓		✓		✓
Uses IMU			✓		✓	✓

Table 4.2: Comparison of COLLAR and related work

multi-user scenario of COLLAR. Finally, LensCap [53] tackles a different problem of AR privacy using fine-grained user permissions, and COLLAR is complementary in that it uses the permitted network data to enable multi-user AR.

Multi-user AR: CARS [125] shares hologram coordinates between users based on coordinates of real-world objects, but relies on cloud processing. CollabAR [78] similarly relies on edge processing. AVR [87] and SPAR [94] are closer to this work in that they use SLAM for localization. AVR shares sparse point clouds between multiple vehicles. SPAR shares environment data between multiple mobile devices, but assumes that the holograms location are given in advance. Neither of these works measure energy consumption; in AVR's vehicular scenario, for example, AR computation energy is not a major concern due to plentiful on-board power sources. In contrast, COLLAR focuses on mobile AR in which energy limitations are a concern.

Past work on wireless sensor networks [25, 29] saves energy by deciding which nodes should sleep/wake up to cover the entire network; such algorithms could be adapted for multi-user AR to select the primary, in future work.

Localization: SLAM-based localization is a key component for off-the-shelf AR systems [43, 14, 82] to enable accurate sharing of hologram positioning information. Edge-SLAM [19] relies on edge infrastructure to speed up SLAM processing, unlike our infrastructure-less approach. Research on multi-user SLAM [130, 2] neglect the AR aspects such as hologram positioning and DNN executions. COLLAR builds on SLAM techniques from the robotics community [70], but adds multi-user capabilities with energy savings.

4.8 Scope and Future Work

COLLAR assumes that the trajectories of the primary and secondary devices overlap at some instance(s) in time, so that they can perform coordinate system synchronization. It also assumes that the primary is able to view all the objects of interest relevant to the AR experience. However, one may envision a relaxation of these assumptions for a more comprehensive AR system. For example, a subset of devices could be assigned as primaries, with the remaining devices as secondaries, based on the set of objects visible in each device's FoV. We also note that COLLARcan use MARLIN's optimizations on the primary to further save power. We plan to address such issues in future work.

4.9 Conclusions

Our work set out to answer a question that applies to many practical settings: Can we enable a rich AR experience in infrastructure-less settings, running natively on user devices, without significant energy drain? Our system COLLARis proof that this elusive goal can be within our reach. The key enabler is a high-level idea of collaborative time slicing to reuse/reduce compute heavy tasks such as DNN/SLAM. While conceptually simple, this entails tackling key correctness, synchronization, and recovery challenges that arise in decentralized AR operation. We showed that COLLARimproves the power consumption of users by up to 36% compared to state of the art AR systems, while also improving the detection accuracy of objects in the real world by nearly 80%. COLLARthus can enable (perhaps for the first time) an infrastructure-free low power framework that can allow users to engage in an AR experience on the fly!

Chapter 5

Conclusions

In this dissertation, we emphasize on the importance of latency and energy issues for mobile augmented reality. For the former, we discover factors contributing to long latency for multi-user AR over cellular networks, especially public LTE. We propose optimizations for AR that are shown to be effective on LTE, and we argue they are applicable for 5G cellular networks as well. For the latter, we propose low-power systems for single and multi-user AR. By intelligently controlling heavy weight computations such as DNN and SLAM, we show that the vision for low-power AR is within our reach.

Bibliography

[1] Fuad Abazovic. Qualcomm snapdragon 835 does not throttle. `https://www.fudzilla.com/reviews/43194-qualcomm-snapdragon-835-does-not-throttle`, 2017.

[2] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. Carmap: Fast 3d feature map updates for automobiles. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 1063–1081, 2020.

[3] Alexandre Alahi, David Marimon, Michel Bierlaire, and Murat Kunt. A master-slave approach for object detection and matching with fixed and mobile cameras. In *2008 15th IEEE International Conference on Image Processing*, pages 1712–1715, 2008.

[4] Mehdi Alasti, Behnam Neekzad, Jie Hui, and Rath Vannithamby. Quality of service in wimax and lte networks. *IEEE Communications Magazine*, 48(5):104–111, 2010.

[5] 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.

[6] Mohmmad Anas, Claudio Rosa, Francesco D Calabrese, Klaus I Pedersen, and Preben E Mogensen. Combined admission control and scheduling for qos differentiation in lte uplink. In *IEEE Vehicular Technology Conference*, 2008.

[7] Android. Android sensor: Linear acceleration. `https://developer.android.com/reference/android/hardware/Sensor\#TYPE_LINEAR_ACCELERATION`.

[8] Android. Wi-fi direct (peer-to-peer or p2p) overview. `https://developer.android.com/guide/topics/connectivity/wifip2p`.

[9] Anonymous. Collar webpage. `https://sites.google.com/view/collar-ar/`.

[10] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury. Marlin demo site. `https://sites.google.com/view/marlin-ar/home`, 2019.

[11] Kittipat Apicharttrisorn, Bharath Balasubramanian, Jiasi Chen, Rajarajan Sivaraj, Yi-Zhen Tsai, Rittwik Jana, Srikanth V. Krishnamurthy, Tuyen Tran, and Yu Zhou.

Custom mobileinsight analyzer and python scripts for ran-level deep analysis. https://github.com/patrick-ucr/ran_latency_analyer_mi.

[12] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V. Krishnamurthy, and Amit K. Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. *ACM SenSys*, 2019.

[13] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V. Krishnamurthy, and Amit K. Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Conference on Embedded Networked Sensor Systems*, SenSys, 2019.

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

[15] Apple. Arkit - apple developer. https://developer.apple.com/arkit/, 2019.

[16] ARToolKit. Artoolkit. http://www.hitl.washington.edu/artoolkit/.

[17] John L Barron, David J Fleet, and Steven S Beauchemin. Performance of optical flow techniques. *International journal of computer vision*, 12(1):43–77, 1994.

[18] Kayce Basques. Get started with remote debugging android devices. https://developers.google.com/web/tools/chrome-devtools/remote-debugging.

[19] Ali J. Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. Edge-slam: Edge-assisted visual simultaneous localization and mapping. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, New York, NY, USA, 2020. ACM.

[20] S. Benhimane and E. Malis. Real-time image-based tracking of planes using efficient second-order minimization. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[22] Boost. Boost library. https://www.boost.org/.

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

[24] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. Mobirnn: Efficient recurrent neural network execution on mobile gpu. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*. ACM, 2017.

[25] Mihaela Cardei and Jie Wu. Coverage in wireless sensor networks. *Handbook of sensor networks*, 21:201–202, 2004.

[26] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *Computer Vision and Pattern Recognition (CVPR)*, pages 4724–4733. IEEE, 2017.

[27] L. Cehovin, A. Leonardis, and M. Kristan. Visual object tracking performance measures revisited. *IEEE Transactions on Image Processing*, 25(3):1261–1274, March 2016.

[28] 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.

[29] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *Wireless networks*, 8(5):481–494, 2002.

[30] 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. In *Conference on Embedded Networked Sensor Systems (SenSys)*, pages 292–304. ACM, 2018.

[31] 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.

[32] 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 *ACM/IEEE Symposium on Edge Computing*. ACM, 2017.

[33] L. Dan, J. Dai-Hong, B. Rong, S. Jin-Ping, Z. Wen-Jing, and W. Chao. Moving object tracking method based on improved lucas-kanade sparse optical flow algorithm. In *2017 International Smart Cities Conference (ISC2)*, pages 1–5, Sept 2017.

[34] 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.

[35] Mark Dennison and Theron Trout. The accelerated user reasoning for operations, research, and analysis (aurora) cross-reality common operating picture (technical report). In *Army Research Laboratory*, 2020.

[36] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, Jun 2010.

[37] Dorian Galvez-López and Juan D. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 2012.

[38] 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.

[39] Google. Arcore cloud anchor api. `https://console.cloud.google.com/marketplace/details/google/arcorecloudanchor.googleapis.com`.

[40] Google. Google just a line. `https://justaline.withgoogle.com/`.

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

[42] Google. Android marshmallow 3.10 scheduler. `https://android.googlesource.com/kernel/msm/+/android-msm-bullhead-3.10-marshmallow-dr/Documentation/scheduler/sched-hmp.txt`, 2015.

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

[44] Google. Arcore overview. `https://developers.google.com/ar/discover/`, 2019.

[45] Google. AugmentedImageActivity.java. `https://github.com/google-ar/arcore-android-sdk/blob/master/samples/augmented_image_java/app/src/main/java/com/google/ar/core/examples/java/augmentedimage/AugmentedImageActivity.java`, 2019.

[46] Google. Google cloud tpu. `https://cloud.google.com/tpu/`, 2019.

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

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

[49] 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. *ACM Mobisys*, 2016.

[50] 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.

[51] 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/`, 2016.

[52] 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.

[53] Jinhan Hu, Andrei Iosifescu, and Robert LiKamWa. Lenscap: split-process framework for fine-grained visual privacy control for augmented reality apps. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 14–27, 2021.

[54] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95. ACM, 2017.

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

[56] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *IEEE conference on computer vision and pattern recognition (CVPR)*, volume 2, page 6, 2017.

[57] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

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

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

[60] Hung-Chin Jang and Chien-Piao Hu. Fairness-based adaptive qos scheduling for lte. In *IEEE International Conference on ICT Convergence*, 2013.

[61] 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.

[62] Kaleb. Fsensor. `https://github.com/KalebKE/FSensor`.

[63] 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.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[65] Kashish Kumawat. Snapdragon 835 review and benchmark test. `https://www.techcenturion.com/snapdragon-835`, 2017.

[66] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *International Conference on Information Processing in Sensor Networks*, IPSN, pages 23:1–23:12, Piscataway, NJ, USA, 2016. IEEE Press.

[67] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, and et al. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017.

[68] Steven LaValle. *Virtual Reality.* Cambridge University Press, 2016.

[69] Magic Leap. Magic leap. `https://www.magicleap.com/`.

[70] Peiliang Li, Tong Qin, Botao Hu, Fengyuan Zhu, and Shaojie Shen. Monocular visual-inertial state estimation for mobile augmented reality. In *2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2017.

[71] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *ACM MobiCom*, 2016.

[72] Pengpeng Liang, Yifan Wu, Hu Lu, Liming Wang, Chunyuan Liao, and Haibin Ling. Planar object tracking in the wild: A benchmark. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[73] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *International conference on Mobile systems, applications, and services (MobiSys)*, pages 69–82. ACM, 2013.

[74] I. Lin, B. Jeff, and I. Rickard. Arm platform for performance and power efficiency - hardware and software perspectives. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–5, April 2016.

[75] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

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

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

[78] 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 *2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2020.

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

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

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

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

[83] 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.

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

[85] Noboru Noguchi, Jeff Will, John Reid, and Qin Zhang. Development of a master–slave robot system for farm operations. *Computers and Electronics in Agriculture*, 2004.

[86] Alexander Pacha. Sensor fusion demo. `https://github.com/apacha/sensor-fusion-demo`.

[87] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–95, 2018.

[88] Qualcomm. Snapdragon power optimization sdk. `https://developer.qualcomm.com/software/snapdragon-power-optimization-sdk`, 2019.

[89] Qualcomm. Snapdragon xr1 platform. `https://www.qualcomm.com/products/snapdragon-xr1-platform`, 2019.

[90] Qualcomm. Trepn power profiler. `https://developer.qualcomm.com/software/trepn-power-profiler`, 2019.

[91] 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.

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

[93] Xukan Ran, Carter Slocum, Maria Gorlatova, and Jiasi Chen. ShareAR: Commnication-Efficient Mobile Augmented Reality. *ACM HotNets*, 2019.

[94] Xukan Ran, Carter Slocum, Yi-Zhen Tsai, Kittipat Apicharttrisorn, Maria Gorlatova, and Jiasi Chen. Multi-user augmented reality with communication efficient and spatially consistent virtual objects. In *ACM CoNEXT*, pages 386–398, 2020.

[95] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *IEEE CVPR*, 2017.

[96] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[97] Matthew Reynolds. Pokemon go buddy adventure explained - how to get hearts, excited buddies, and all buddy level rewards including best buddy explained. `https://www.eurogamer.net/articles/2019-12-19-pokemon-go-buddy-adventure-play-excited-6002`.

[98] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *International Conference on Computer Vision*, ICCV, pages 2564–2571. IEEE, 2011.

[99] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[100] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

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

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

[103] Sheng Shen, Mahanth Gowda, and Romit Roy Choudhury. Closing the gaps in inertial motion tracking. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, New York, NY, USA, 2018. ACM.

[104] Arno Solin, Santiago Cortes, Esa Rahtu, and Juho Kannala. Inertial odometry on handheld smartphones. In *2018 21st International Conference on Information Fusion (FUSION)*, 2018.

[105] Mirko Suznjevic, Jose Saldana, Maja Matijasevic, Julián Fernández-Navajas, and José Ruiz-Mas. Analyzing the Effect of TCP and Server Population on Massively Multiplayer Games. *International Journal of Computer Games Technology*, 2014(602403):1 – 17, 2014.

[106] Richard Szeliski. *Computer vision: algorithms and applications.* Springer Science & Business Media, 2010.

[107] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[108] Zhenjun Tang, Y. Dai, and X. Zhang. Perceptual hashing for color images using invariant moments. *Applied Mathematics and Information Sciences*, 6:643S–650S, 04 2012.

[109] TensorFlow. Tensorflow lite object detection example. `https://www.tensorflow.org/lite/examples/object_detection/overview`.

[110] TensorFlow.org. Tensorflow android camera demo. `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android`.

[111] TensorFlow.org. Ssdlite mobilenets v2 config file. `https://github.com/tensorflow/models/blob/master/research/object\_detection/samples/configs/ssdlite\_mobilenet\_v2\_coco.config`, 2018.

[112] TensorFlow.org. Tensorflow lite gpu delegate. `https://www.tensorflow.org/lite/performance/gpu`, 2019.

[113] TensorFlow.org. Tensorflow lite. `https://www.tensorflow.org/lite`, 2021.

[114] Ramona Trestian, Arghir-Nicolae Moldovan, Olga Ormond, and Gabriel-Miro Muntean. Energy consumption analysis of video streaming to android mobile devices. In *2012 IEEE Network Operations and Management Symposium*, 2012.

[115] Ana Villanueva, Zhengzhe Zhu, Ziyi Liu, Kylie Peppler, Thomas Redick, and Karthik Ramani. Meta-ar-app: An authoring platform for collaborative augmented reality in stem classrooms. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 2020.

[116] Vuforia. Augmented reality for the industrial enterprise. `https://www.vuforia.com/`, 2019.

[117] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. Real-time detection and tracking for augmented reality on mobile phones. *IEEE transactions on visualization and computer graphics*, 16(3):355–368, 2010.

[118] Liming Wang, Jianbo Shi, Gang Song, and I-Fan Shen. Object detection combining recognition and segmentation. In *Asian Conference on Computer Vision*, ACCV, pages 189–199, Berlin, Heidelberg, 2007. Springer-Verlag.

[119] X. Wang, M. Yang, S. Zhu, and Y. Lin. Regionlets for generic object detection. In *IEEE International Conference on Computer Vision (ICCV)*, pages 17–24, Dec 2013.

[120] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E. Gonzalez. IDK cascades: Fast deep learning by learning not to overthink. *CoRR*, abs/1706.00885, 2017.

[121] S. Wu, Y. Fan, S. Zheng, and H. Yang. Object tracking based on orb and temporal-spacial constraint. In *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*, pages 597–600, Oct 2012.

[122] Zongkai Wu and Wei Wang. Magnetometer and gyroscope calibration method with level rotation. *Sensors*, 2018.

[123] Z. Yang. Fast template matching based on normalized cross correlation with centroid bounding. In *International Conference on Measuring Technology and Mechatronics Automation*, volume 2, pages 224–227, March 2010.

[124] Wenxiao Zhang, Bo Han, and Pan Hui. Jaguar: Low latency mobile augmented reality with flexible tracking. In *International Conference on Multimedia*, pages 355–363. ACM, 2018.

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

[126] Liang Zheng, Yi Yang, and Qi Tian. Sift meets cnn: A decade survey of instance retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 40(5):1224–1244, 2018.

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

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

[129] 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.

[130] 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.

[131] Longhao Zou, Ali Javed, and Gabriel-Miro Muntean. Smart mobile device power consumption measurement for video streaming in wireless environments: Wifi vs. lte. In *2017 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6, 2017.