MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency

Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, Randy H. Katz
Computer Science, UC Berkeley
{kaifei,sasrwas,hs.kim,culler,randykatz}@berkeley.edu

ABSTRACT

This paper presents MARVEL, a mobile augmented reality (MAR) system which provides a notation display service with imperceptible latency (<100 ms) and low energy consumption on regular mobile devices. In contrast to conventional MAR systems, which recognize objects using image-based computations performed in the cloud, MARVEL mainly utilizes a mobile device's local inertial sensors for recognizing and tracking multiple objects, while computing local optical flow and offloading images only when necessary. We propose a system architecture which uses local inertial tracking, local optical flow, and visual tracking in the cloud synergistically. On top of that, we investigate how to minimize the overhead for image computation and offloading. We have implemented and deployed a holistic prototype system in a commercial building and evaluate MARVEL's performance. The efficient use of a mobile device's capabilities lowers latency and energy consumption without sacrificing accuracy.

CCS CONCEPTS

• **Human-centered computing → Mixed / augmented reality**; **Mobile computing**;

KEYWORDS

Mobile Augmented Reality, Selective Offloading, Visual-Inertial SLAM, Mobile Computing

ACM Reference Format:
Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, Randy H. Katz. 2018. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. In *The 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), November 4–7, 2018, Shenzhen, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274783.3274834

1 INTRODUCTION

Augmented Reality (AR) has seen wide adoption with recent advances in computer vision and robotics. The basic concept of AR is to show a user additional information overlaid on the original view when she sees the real world through the screen. A user expects to see the augmented information both accurately and in real time, while moving her device. To this end, an AR device must perform intense computations on a large amount of (visual) data with *imperceptible latency*, from capturing an image to extracting the corresponding information (e.g., image localization and surface detection) and overlaying it on the proper location of the screen.

While this challenge has been addressed by using powerful and expensive AR-oriented hardware platforms, such as Microsoft HoloLens [22] and Google Tango [45], another approach also has been investigated: AR with *regular mobile devices*, such as a smartphone, called Mobile AR (MAR). MAR is attractive in that it does not require users to purchase and carry additional devices [14]. However, MAR has its own challenges since mobile devices have significantly less *storage* and *computation* than AR-oriented devices. In addition, given that mobile platforms are not only for MAR but also for many other daily tasks (e.g., social network applications and web search), MAR is expected to avoid excessive *battery* consumption. There are relatively lightweight MAR platforms that operate only in a small area (e.g., a room), such as Google ARCore [2] and Apple ARKit [4], which can only run on the latest generation of smartphones.

This work investigates how to enable real-time MAR on ordinary mobile devices. Specifically, we aim to provide an *annotation-based MAR service* which identifies objects that the device points at and overlays the corresponding annotations on the relevant location of the screen in real time. This application is viable even with the restricted capabilities of mobile devices in a large area (e.g., a building), in contrast to more demanding AR that naturally embeds 3D objects into a 3D model of a small area, such as ARCore [2], ARKit [4], and Hololens [22]. More importantly, it has a variety of practical usage scenarios, such as museum guides, smart buildings, and airport navigation [15, 26]. Through the annotation service, a user can obtain information about an object by pointing at it. Furthermore, she is able to interact with (or control) the object by touching annotations on the screen.

A number of MAR studies have attempted to implement these applications. They offload computation and storage to the *cloud* to overcome the restrictions of mobile devices. However, some work [15, 26] excessively depends on the cloud, with a mobile device simply uploading large volumes of images, resulting in high latency and energy consumption. Despite some effort to reduce offloading latency [27, 49], the lowest latency is still 250 ms [48] to the best of our knowledge, which is 2.5x higher than the 100 ms requirement for a smooth user interface, based on years of research in Human-Computer Interaction (HCI) [10, 11, 34, 37]. Some work enables fast tracking of an *identified* object on the screen using local optical flow [23], but the identification procedure when new objects appear still relies on the cloud [16, 49]. In addition, optical flow only works with clear images that have sufficient overlap, which is suitable for applications with slow rotating cameras, such

SenSys '18, November 4–7, 2018, Shenzhen, China

ACM ISBN 978-1-4503-5952-8/18/11…$15.00
https://doi.org/10.1145/3274783.3274834

as object tracking in car dash cameras [16]. Furthermore, energy consumption has been significantly overlooked in this regime.

To overcome these challenges, we design MARVEL (MAR with Viable Energy and Latency), which fully utilizes the potential of both the cloud and the mobile device. To identify and track objects on the screen, MARVEL uses visual and inertial 6-DOF (Degree of Freedom) localization, rather than image retrieval and a convolutional neural network (CNN) in previous MAR work. With 6-DOF locations, it can facilitate cooperation between the cloud and the mobile device by synchronizing their coordinate systems and can distinguish instances of the same type. MARVEL mainly relies on light and fast *local inertial data processing* while using heavy and slow local optical flow and cloud offloading *selectively*; it locally computes optical flow only when the device accumulates enough position changes and offloads images only when local results need to be calibrated. This significantly reduces latency, resulting in <100 ms for both object identification and tracking. Furthermore, 6-DOF inertial and image localization allows MARVEL to project *multiple* 3D object labels onto the screen very efficiently with a simple matrix multiplication, allowing us to track existing as well as newly/partially visible objects with little overhead.

With this low-latency system architecture, we aim to minimize energy consumption on the mobile device (both computation and offloading overhead) without sacrificing accuracy. To this end, we explore several essential issues: (1) when to offload, (2) what to offload, and (3) what to do locally while not offloading. MARVEL performs image offloading-based calibration only when it detects an inconsistency between inertial and visual data (optical flow). For image offloading, the mobile device selects only a few recently captured images with two criteria: sharpness and the number of features (more features imply that the image contains more information). We obtain these two metrics not by heavy image computation but indirectly from simple gyroscope readings and edge detection [24]. When not offloading, the mobile device uses inertial data for the MAR service and corrects its errors by processing visual data selectively. Meanwhile, it continuously monitors the inconsistency between the inertial and visual information.

MARVEL's performance is evaluated in a holistic system, which implements a real-time appliance annotation service in a commercial building. The results verify that the above design choices enable MARVEL to achieve high accuracy, low latency, and low energy consumption together.

The contributions of this work are summarized as follows:

- We realize an annotation-based AR service on ordinary mobile devices with *imperceptible end-to-end latency* for a physical world change to be reflected on the device screen (i.e., <100 ms based on research in HCI [10, 11, 34]), the first practical *real-time* MAR system to the best of our knowledge.
- We propose *a novel MAR system architecture* that synergistically bridges image localization in the cloud and inertial localization on a mobile device by sharing their 3D coordinate systems, which decouples network latency from user experience while correcting local errors using the accurate results from the cloud.
- We investigate how to efficiently monitor consistency in local tracking and calibrate locations with cloud offloading. Specifically, we answer the following questions: (1) when to offload, (2) what to offload, and (3) what to do while not offloading.

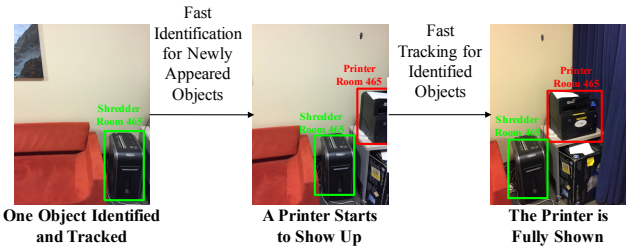


Figure 1: Target application: Real-time, multi-object annotation service on a regular mobile device.

- We implement and deploy *a holistic system*, where MARVEL's performance is evaluated and compared with state-of-the-art techniques, providing a better understanding of MAR system operations.

2 ANNOTATION-BASED MAR SYSTEMS

This section describes the motivation of MARVEL. We first describe the application scenario and technical requirements of annotation-based MAR systems. Then we introduce possible design options and show our preliminary study which motivates our design choices.

2.1 Desired User Experience

We aim to provide an *annotation-based MAR service*, where a regular mobile device identifies objects that it points at and overlays their annotations on the relevant location of the screen in real time. This application should be viable with the restricted capabilities of commercial off-the-shelf mobile devices. Furthermore, it has a variety of practical usage scenarios [15, 26]. For example, when going on a field trip to a museum, a user may see the description of each exhibit simply by pointing the mobile device at it. In a building, she may get a control interface for an appliance, such as a projector on the ceiling or a printer, by pointing at it.

Figure 1 depicts the desired user experience in this application scenario. A user expects the five following benefits, which introduce technical challenges for a practical MAR system design.

- **Fast identification:** When pointing at a new object, she expects its annotation to *immediately* pop up on the relevant location of the screen. She wants to see the annotation even before the new object is fully captured on the screen (i.e., the second case of Figure 1). To this end, an object identification process should be finished within 100 ms.[1]
- **Fast tracking:** After the annotation pops up, she expects it to *smoothly* move *along with* the corresponding object while moving her device. To this end, whenever the image on the screen changes, the annotation view should be updated within 100 ms, as discussed above.
- **Multi-object support:** When multiple objects are being tracked on the screen, she expects fast identification and tracking for each object simultaneously (i.e., the second and the third cases of Figure 1). To this end, the latency of annotation placement process should be decoupled from the number of objects.

[1] Decades of HCI research [10, 11, 34] have shown that 100 ms is "about the limit for having the user feel that the system is reacting instantaneously," as described in the book "Usability Engineering" [37]. As we will show in Section 6, the latency for an Android camera app to reflect a physical world change is also close to 100 ms.

- **Robustness to arbitrary movement:** The user does not have to intentionally move her device slowly to use this service. To this end, the accuracy of annotation placement should be consistent regardless of device movement.
- **Low energy consumption:** Her device is a regular smartphone, used not only for this application but also for other various purposes. She does not want to recharge the device at an excessively higher frequency if using this application.

2.2 Design Options

To fulfill the above requirements, system design choices need to be made carefully. It is important to decide how to recognize and track objects and where to run core algorithms, which impact accuracy, latency, and energy consumption.

How to Recognize and Track Objects? In our application scenario, we are not only recognizing the categories of objects, but also *instances*. For example, two printers of the same model in two rooms need to be differentiated based on the contextual information. There are three representative ways to recognize instances and track objects: image retrieval [26], Convolutional Neural Networks (CNN) [50], and 6-DOF localization [15]. With a query image on the screen, image retrieval selects the most similar labeled image in the database in terms of the number of common salient features (e.g., SURF [6]), while a CNN extracts features from the query image and uses them to classify the image on a pre-trained neural network. These require a large database and heavy computation [15]. Even with recent work on running image retrieval or CNNs on smartphones [20, 28], they require an on-board GPU and are not still energy efficient.

On the other hand, accurate 6-DOF locations can be used to identify and track an object by projecting labeled 3D points onto the image plane [15]. Various localization techniques have been developed for decades, which use different types of information (e.g., 2D image, depth image, LIDAR scans, and inertial data) and have different accuracy, latency, and storage requirements. For example, visual simultaneous localization and mapping (VSLAM) [29] is accurate but extremely computationally heavy, while inertial localization is lightweight but more error-prone [43]. *We choose 6-DOF location-based object recognition and tracking.* Our intuition is that a synergistic combination of heterogeneous localization techniques has the potential to achieve the above five requirements by overcoming the weakness of each technique.

Where to Perform Computation? Another question is how to efficiently use the capabilities of a cloud server and a mobile device [8]. Offloading heavy computation to the cloud reduces computation overhead on a mobile device. However, it introduces energy consumption and latency costs from communication instead of computation. Therefore, it is important to decide what algorithms to compute locally or in the cloud, what information to offload, and when to offload, as these factors significantly impact our performance.

When making these design choices, a localization-based system provides more options with diverse techniques than other options. However, each localization technique's characteristics need to be carefully analyzed in order to make proper design decisions.

Operation	Power (Watt)
Preview only	1.96 ± 0.64
Local inertial localization	1.94 ± 0.74
Local image localization [29]	5.74 ± 2.44
Offloaded image localization	2.24 ± 0.68
Optical flow	2.54 ± 1.06

Table 1: Power usage of different applications. Offloaded image localization and local optical flow consume more power than local inertial localization. Local image localization (RTABMap [29]) consumes 2.5× more power than offloaded image localization.

Operation	Time (ms)
Inertial (Rotation)	0.01 ± 0.03
Inertial (Translation)	0.03 ± 0.08
Optical flow	30.68 ± 5.16
Image offloading	≥ 250 [48]

Table 2: Processing time of different operations. Local optical flow and image offloading take significantly more time than operations in local inertial localization.

2.3 Preliminary Study

To design a location-based MAR system satisfying the five requirements in Section 2.1, we evaluate the power usage and computation time of some candidates: (1) local inertial localization, (2) local image localization, (3) image localization offloaded to the cloud, and (4) local optical flow, which is not localization but commonly used in other MAR systems [16].

We run a preview app (i.e., a see-through camera) as a baseline, and four applications corresponding to the above cases on a Lenovo Phab 2 Pro Android smartphone: (1) a preview app that performs 6-DOF inertial localization in the background (translation and rotation), (2) RTABMap [29], a purely local image localization application, which can be considered equivalent to ARCore [2] or ARKit [4] without 3D object rendering, (3) a preview app that offloads an image with resolution 640×360 to the cloud for 6-DOF image localization sequentially (i.e., it only offloads the next image when the current image result comes back), (4) a preview app that performs local optical flow tracking on sequential images with resolution 640×360.[2] We measure the system power usage using the Qualcomm Trepn Power Profiler [46] for accurate battery usage, enabled by the Qualcomm Snapdragon CPU in the smartphone.

Table 1 shows the power consumption of the applications. First, local image localization consumes much more energy than the other applications. In addition, we cannot store a 3D model at the scale of a building on a smartphone. This confirms that using the cloud is necessary for MAR. Second, both image offloading and optical flow consume significant energy, which reveals that both need to be avoided whenever possible. Lastly, local inertial localization does not add noticeable energy overhead compared to the basic preview app. This is consistent to results in prior work [13], which shows that inertial measurement unit (IMU) consumes at most ~30 mW.

[2]Note that our 3D model is constructed from images with resolution 640×480. Using images with higher resolution improves accuracy marginally but significantly increases computation and transmission latency.

After eliminating the local image localization option (the most expensive approach), we measure the processing time of relevant operations in the remaining applications, as shown in Table 2. Since network performance varies from place to place, we use the data summarized in [48] for offloading latency. As we can see, processing inertial data (both rotation and translation) is orders of magnitude faster than optical flow and image offloading. In addition, offloading an image not only consumes more energy, but also takes longer than one optical flow computation.

These results confirm that local computation of 6-DOF inertial localization is the fastest and the most energy-efficient localization method for MAR. However, given that inertial localization is error-prone [43], the question is how to compensate for its errors while maintaining energy efficiency. Our idea is to use both local optical flow and image localization in the cloud to detect errors and improve accuracy. At the same time, we try to trigger these expensive methods *only when necessary* to minimize energy consumption on the mobile device.

3 MARVEL OVERVIEW

This section gives an overview of MARVEL, a new *localization*-based MAR system for real-time annotation services which fulfills the five application requirements discussed in Section 2.1.

As a *holistic* MAR system, the primary contribution of MARVEL is its comprehensive system architecture. While it uses existing algorithms, such as 6-DOF inertial localization with Zero Velocity Update (ZUPT) [43], 6-DOF image localization [40], and optical flow [23], MARVEL's design focuses on how to combine these algorithms as well as when and where to run each algorithm considering both cloud offloading and local computing. Each design choice has a significant impact on MAR performance and a synergistic system design is the key factor to achieve both low energy and low latency.

To generate an annotation view, MARVEL first obtains the mobile device's 6-DOF location and uses it to obtain 2D locations of objects on the screen. While inertial localization is lightweight but accurate only for a short time [31], image localization is accurate but computationally heavy and involves a large volume of data. Our MARVEL system achieves the sweet spot of this trade-off by focusing on the following design aspects:

(1) Database: MARVEL constructs a heavy database (a 3D point cloud) for image localization in the cloud, while constructing a *lightweight* local database on the mobile client to enable local object identification and tracking (Section 4.1).

(2) Local computing: The mobile client mainly performs inertial localization, identifying and tracking objects and generating an annotation view, *all by itself* using its lightweight local database. This excludes network latency from the user experience, resulting in real-time MAR. In addition, it uses optical flow [23] (visual data processing) together with the inertial localization results to improve accuracy of the annotation location and detect inconsistency (triggering calibration in the cloud). Optical flow is performed *only when necessary*, minimizing local image computation while maintaining high accuracy (Sections 4.2 to 4.4).

(3) Cloud offloading: The mobile client triggers image localization in the cloud only when IMU-based results need calibration (when detecting inconsistency) to maintain accuracy with minimal offloading overhead. When triggering a calibration, it selects *only a few* images to offload and the image selection process is also *lightweight*, minimizing both communication and computation overhead (Sections 4.5 and 4.6).

Overall, the main idea of MARVEL is to primarily use local computations involving IMU data while triggering heavy image computation (optical flow) and offloading (image localization in the cloud) only when necessary. This design choice aims to achieve low latency and low energy consumption without sacrificing accuracy.

4 SYSTEM DESIGN

Figure 2 illustrates MARVEL's architecture and operational flow, which involves a mobile client (carried by a user) and a cloud server. The client has a local database and performs AR locally most of the time with 6-DOF inertial localization and optical flow. When necessary, the client also communicates with the cloud server to calibrate its inertial localization results. To perform calibration, the cloud server has a pre-built 3D point cloud database and provides 6-DOF image localization after receiving a query image from the client.

4.1 Initialization: Database Construction

Initially, the point cloud of the complete service area (e.g., a building), along with all annotated 3D points, is created and stored in the cloud server as the *MARVEL database*. We construct the 3D point cloud database as in [15], which is easier than constructing a database for image retrieval or CNN. Specifically, we take a 2D video stream by using an existing SfM tool [29] while walking through each room of the target building (one video per room, multiple videos per building). Then the SfM tool computes the relative 6-DOF location between every two consecutive images in the video stream (of a room) by matching their salient features (e.g., SURF [6]) and projects all the 2D feature points into a 3D coordinate system (a.k.a. frame), denoted as the model frame $\mathcal{M}$ in Figure 3. In the constructed 3D point cloud, we manually label each object *only once*, as a 3D labeled point.

We denote the frame of an image captured by the client at time t as the image frame $\mathcal{I}(t)$ in Figure 3. The image frame is the same as the frame of device screen and changes over time as the device moves. When the client uploads this image to the cloud for image localization, the returned 6-DOF location is denoted as $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t)}$. Note that $\mathbf{P}^{\mathcal{A}}_{\mathcal{B}}$ is the 4×4 homogeneous transformation from frame $\mathcal{A}$ to frame $\mathcal{B}$, represented as

$$\mathbf{P}^{\mathcal{A}}_{\mathcal{B}} = \left(\begin{array}{ccc|c} & \mathbf{R}^{\mathcal{A}}_{\mathcal{B}} & & \mathbf{T}^{\mathcal{A}}_{\mathcal{B}} \\ 0 & 0 & 0 & 1 \end{array} \right) \tag{1}$$

where $\mathbf{R}^{\mathcal{A}}_{\mathcal{B}}$ and $\mathbf{T}^{\mathcal{A}}_{\mathcal{B}}$ are the 3×3 rotation matrix and the 3×1 translation vector from frame $\mathcal{A}$ to $\mathcal{B}$, respectively.

Unlike previous MAR systems that maintain all of their data only in the cloud and completely rely on cloud computing, MARVEL efficiently utilizes local computing. To this end, while storing the heavy 3D point cloud database in the cloud, MARVEL also constructs a *lightweight* local database in the mobile client. Specifically, when a user enters the service area (e.g., building), the mobile client downloads the 3D-location and annotation of each labeled object from the

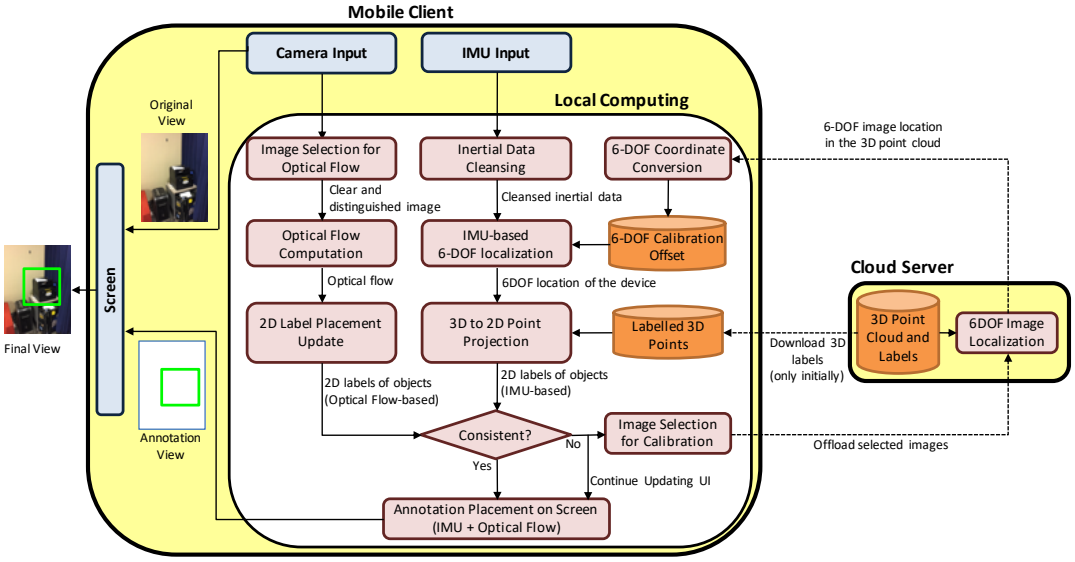


Figure 2: MARVEL system operation overview. There are a mobile client and a cloud server. The local computing module on the mobile client takes the main role for generating annotation view by using inertial localization and optical flow. The cloud does image localization to calibrate local tracking errors, which is triggered selectively.

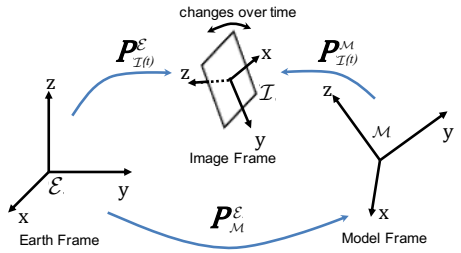


Figure 3: Three frames for 6-DOF localization with inertial and visual Data: earth frame $\mathcal{E}$, model frame $\mathcal{M}$, and image frame $\mathcal{I}(t)$. While $\mathcal{E}$ and $\mathcal{M}$ are static, $\mathcal{I}(t)$ changes over time as the device screen moves.

cloud database.[3] Each downloaded label L has its annotation A^L and its 3D position in the model frame $\mathcal{M}$, $\mathbf{p}^L_{\mathcal{M}} = (x^L_{\mathcal{M}}, y^L_{\mathcal{M}}, z^L_{\mathcal{M}})^T$. This local database requires very small storage and communication overhead. Assuming that the average annotation length is 100 bytes, an entire building that contains 10,000 labels will require only 1.12 MB ($= 10,000 \times (3 * 4Bytes + 100Bytes)$) of local storage. Furthermore, this label downloading happens *very rarely* (e.g., when entering into another building), incurring negligible communication overhead.

Given that local 6-DOF inertial localization is performed in the earth frame $\mathcal{E}$ and returns $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t)}$, each object's 3D location needs to be represented in the earth frame $\mathcal{E}$ to enable the mobile client to locally identify an object based on inertial localization. To this end, the mobile client converts the model frame-based 3D position of each downloaded label L, $\mathbf{p}^L_{\mathcal{M}}$, to $\mathbf{p}^L_{\mathcal{E}}$, the 3D position in the earth frame $\mathcal{E}$, by using a constant matrix $\mathbf{P}^{\mathcal{E}}_{\mathcal{M}}$. Note that both $\mathcal{E}$ and $\mathcal{M}$ are static frames and thus $\mathbf{P}^{\mathcal{E}}_{\mathcal{M}}$ is also constant, which is computed once shortly after the MARVEL app starts (when inertial localization is still accurate) and does not change over time. With the local database ($\mathbf{p}^L_{\mathcal{E}}$ and A_L), the mobile client is able to identify an object and puts its annotation on the screen *by itself*.

[3]The client downloads only labels, rather than the whole 3D point cloud.

4.2 Local Inertial Tracking

While MARVEL is running, the mobile client's local computing module continuously receives inertial data (acceleration and rotation) in the earth frame $\mathcal{E}$ and camera feed, and computes the two types of data in parallel. Sections 4.2 through 4.4 present how this parallel local processing of inertial and visual data results in an annotation view overlaid on the device screen, as depicted in Figure 2.

We first consider inertial data processing. After receiving accelerometer data in the earth frame $\mathcal{E}$ from the sensor, the client first cleans this data by zeroing all readings below a threshold (0.2 m/s^2 in all three dimensions in MARVEL). The client then computes its 6-DOF location in the earth frame $\mathcal{E}$ at time t, $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t)}$, as

$$\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t)} = \mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_0)} \cdot \mathbf{P}^{\mathcal{I}(t_0)}_{\mathcal{I}(t)}. \quad (2)$$

Here $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_0)}$ is the *calibration offset* obtained at time t_0 (when the last calibration is triggered, i.e., $t_0 < t$), and $\mathbf{P}^{\mathcal{I}(t_0)}_{\mathcal{I}(t)}$ is the transformation from the image frame $\mathcal{I}(t_0)$ to $\mathcal{I}(t)$, computed using the latest cleaned accelerometer data and the latest rotation sensor data. In addition, we use the Zero Velocity Update (ZUPT) [43] algorithm to correct the location when the user stops moving.

In a calibration event, the client sends a query image (captured at time t_0, with the image frame $\mathcal{I}(t_0)$) to the cloud and receives $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_0)}$, the result of 6-DOF image localization, from the cloud. Then it converts $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_0)}$ to the calibration offset $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_0)}$ (from the model frame $\mathcal{M}$ to the earth frame $\mathcal{E}$) using

$$\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_0)} = \mathbf{P}^{\mathcal{E}}_{\mathcal{M}} \cdot \mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_0)} \quad (3)$$

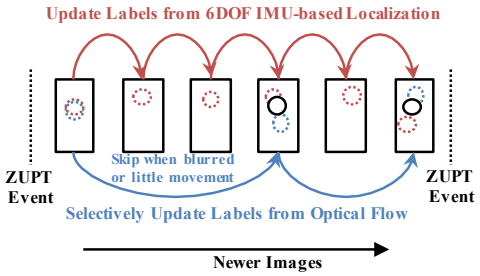


Figure 4: Correcting the results of local inertial tracking with those of selective local visual tracking (optical flow).

where $\mathbf{P}^{\mathcal{E}}_{\mathcal{M}}$ was already computed in the initialization phase as described in Section 4.1. The calibration offset can be used as long as the inertial localization is accurate. As we will discuss in Section 4.5, the calibration is triggered *only when necessary* (when the inertial localization turns out to be inaccurate) to minimize offloading overhead. Note that a calibration takes hundreds of milliseconds to finish since it includes image offloading. While waiting for a calibration to finish, the current, outdated calibration offset is still used. Therefore, a calibration should be triggered early enough considering this delay.

After getting its 6-DOF location information $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t)}$, the mobile client projects 3D-points of nearby objects (i.e., $\mathbf{p}^{L}_{\mathcal{E}}$ in the *local database*) onto the screen [21]. To do this, the client extracts $\mathbf{R}^{\mathcal{E}}_{\mathcal{I}(t)}$ and $\mathbf{T}^{\mathcal{E}}_{\mathcal{I}(t)}$ from $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t)}$, and computes as follows

$$\begin{bmatrix} \mathbf{s}^{L}_{\text{imu}}(t) \\ 1 \end{bmatrix} = \mathbf{K}_{3\times3} \left[\mathbf{R}^{\mathcal{E}}_{\mathcal{I}(t)} \middle| \mathbf{T}^{\mathcal{E}}_{\mathcal{I}(t)} \right] \begin{bmatrix} \mathbf{p}^{L}_{\mathcal{E}} \\ 1 \end{bmatrix} \tag{4}$$

where $\mathbf{K}_{3\times3}$ is a hardware-dependent intrinsic matrix obtained out-of-band once. It allows us to convert a 2D location on the screen plane to its 2D pixel location on the screen. The result $\mathbf{s}^{L}_{\text{imu}}(t)$ is a 2×1 vector that represents IMU-based estimation of 2D pixel location on the screen, for label L at time t. Note that $\mathbf{s}^{L}_{\text{imu}}(t)$ is generated only with local computation because we always have a calibration offset $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_0)}$ available from the previous calibration.

4.3 Selective Local Visual Tracking

While performing local inertial tracking, the mobile client also processes visual data (given by the camera) in parallel. After receiving a new image from the camera, the client first checks if it is sharp enough for accurate optical flow using Sobel edge detection [44], which only takes around 4 ms to compute. If the sum of the detected edge intensities is below a threshold (500, 000 for a 640×360 image in our settings), it does not consider the image for optical flow. If the new image is sharp enough, the client computes optical flow only when the new image is significantly different from the previous image, as depicted in Figure 4. Our idea is that optical flow between two very similar images consumes nontrivial energy (as shown in Section 2.3) without contributing to accuracy improvement, which should be avoided.

How can we measure the difference between two images? The pixel-wise image comparison adopted by previous work on car-mounted cameras [16] does not work well for smartphones, because inconsistent hand movements can cause different levels of blurriness (even among the sharp images). Instead, we indirectly use the location difference provided by the IMU (both rotation and translation). We compare the two IMU data points, each of which is sampled when each of the two images is taken. If the rotation or translation accumulated since the previous optical flow computation is sufficiently large (more than 5° or 1 cm in our settings), the client triggers a new optical flow computation. This results in $\mathbf{s}^{L}_{\text{of}}(t)$, which is optical flow-based estimation of 2D pixel location on the screen, for label L at time t.

4.4 Smoothing Annotation Movement

Now the mobile client has two sets of 2D points (pixel location) for each object on the screen, $\mathbf{s}^{L}_{\text{imu}}(t)$ from inertial localization and $\mathbf{s}^{L}_{\text{of}}(t')$ from optical flow. t is the current time and t' ($< t$) is the last time when optical flow was computed. Again, optical flow computation is not always triggered for energy savings and takes longer than inertial data computation. $\mathbf{s}^{L}_{of}(t')$ is good enough to be used at time t since the client does not move significantly between time t' and t (otherwise an optical flow would have been triggered).

To compensate for the cumulative errors of $\mathbf{s}^{L}_{\text{imu}}(t)$ caused by translation and to achieve smooth movement of annotations on the screen, MARVEL combines $\mathbf{s}^{L}_{\text{imu}}(t)$ and $\mathbf{s}^{L}_{\text{of}}(t')$ to compute $\mathbf{s}^{L}_{\text{final}}(t)$, the final 2D pixel location of label L at time t which determines where to place annotation A_L on the screen. $\mathbf{s}^{L}_{\text{final}}(t)$ is given by

$$\mathbf{s}^{L}_{\text{final}}(t) = \frac{c_{\text{imu}}\mathbf{s}^{L}_{\text{imu}}(t) + c_{\text{of}}\mathbf{s}^{L}_{\text{of}}(t')}{c_{\text{imu}} + c_{\text{of}}} \tag{5}$$

where c_{imu} and c_{of} are the confidence level for $\mathbf{s}^{L}_{\text{imu}}(t)$ and $\mathbf{s}^{L}_{\text{of}}(t')$ respectively.

Given that inertial localization error mainly comes from translation (double integration of linear acceleration) [43], we decrease c_{imu} as the delay between time t_0 (when the last calibration was triggered) and t increases. On the other hand, c_{of} is obtained from the quality of the optical flow performed at t'. We measure the quality as Glimpse [16] does: using standard deviation of feature differences around all labels between the two images. Lastly, the client decides where to put each annotation A_L using Equation 5 and overlays the annotation view on the original view.

The whole process until generating the annotation view is performed *completely locally*. The mobile client does not wait for any response from the cloud since it has all the necessary information (i.e., the local database and the calibration offset), avoiding a network latency for image offloading that can be greater than 250 ms. The latency of MARVEL comes from local inertial data processing and selective optical flow computation.

4.5 Selective Image Offloading

Sections 4.5 and 4.6 present the calibration mechanism of MARVEL. The novelty of the calibration mechanism is in the method of providing an accurate calibration offset without heavy computation and communication overhead on the mobile client.

When the mobile client detects inconsistency while performing the above local computation, it triggers calibration on the cloud server. Inconsistency triggers if one of the following condition happens: (1) when both confidence values, c_{imu} and c_{of}, are zero, or (2) when $\text{dist}(\mathbf{s}^{L}_{\text{imu}}(t), \mathbf{s}^{L}_{\text{of}}(t')) > \phi$ (i.e., 2D points generated by inertial localization and optical flow are significantly different),

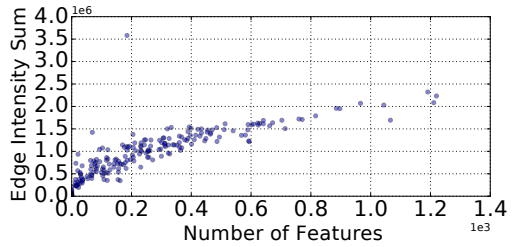


Figure 5: Correlation between the number of features and sum of edge intensity. We use SURF as features and Sobel edge detection, both using the default OpenCV parameters. The experiment is conducted with 207 images with resolution 640×360.

where ϕ is called the *calibration threshold*. As ϕ decreases, accuracy increases but offloading overhead also increases. This trade-off will be investigated in Section 6.2.

If the client decides to trigger a calibration, it offloads images to the cloud. In doing so, it offloads *only a subset of the recently captured images* to reduce communication overhead. The impact of the number of offloading images N will be investigated in Section 6.3. To avoid losing image localization accuracy while offloading only a few images, the client needs to extract each image's features and measure the feature uniqueness [27]. However, this process requires heavy image computation on the client, making computation overhead exceed even the offloading overhead [26], especially when we want to use a robust feature, such as SURF [6].

To avoid this, in MARVEL, the client does not consider feature uniqueness but image sharpness and the number of features when selecting the best query images. Offloading a clear image can help the cloud to extract meaningful features from it. Another intuition is that an image with more features is likely to have more unique features at the same time (which is not always true but good enough as a heuristic). The client infers image sharpness (or blurriness) from both *gyroscope readings* and *edge detection*, because blurry images mostly come from rotations [24], and sharp images tend to show more edges. In addition, images with more edges should have more features, because features are essentially descriptors of high-contrast areas, most of which are on edges. To verify this, we test 207 images captured in a campus building. Figure 5 shows that the sum of edge pixels (given by edge detection) and the number of features of the 207 images are highly correlated.

To select the best images for calibration, the mobile client ranks the images in its cache based on the sum of edge pixels and the gyroscope reading at the time of the image capture, resulting in r_i^{edge} and r_i^{gyro} for image i, respectively (smaller rank is better). Then it ranks the images by $(r_i^{\text{edge}} + r_i^{\text{gyro}})$ and offloads the top N images to the cloud. Overall, this image selection process is much faster and more energy-efficient than computing feature uniqueness directly [27].

4.6 Calibration

Upon receiving the N images, the server performs image localization for all of them, and returns N results to the client: 6-DOF location $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$ for $i \in \{1, 2, \ldots, N\}$, where t_i is the time when image i was captured. Then, the client tries to select the most accurate $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$ among the N results and uses it for the calibration offset. Although $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1)}$ (the result from the best ranked offloading image) usually gives the best calibration offset, the other $N-1$ results still need to be considered as sometimes $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1)}$ can be an outlier (Note that we do not directly consider feature uniqueness when ranking offloading images).

To compare the N results together, the client estimates a $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1)}$ from each $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$. We denote $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ to be the estimation of $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1)}$ derived from $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$, which can be obtained by using inertial localization results:

$$\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)} = \mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)} \left(\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_i)}\right)^{-1} \mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_1)} \tag{6}$$

If both the image localization and inertial localization are ideal, $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ is the same for all i in $\in \{1, 2, \ldots, N\}$. In practice, both localization methods have errors, which causes differences among them.

When selecting the best result, our intuition is that if $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ is similar to the $N-1$ others, $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$ should be safe to use for the calibration offset (i.e., it has a lower chance to be an outlier). To measure the similarity, the client gets $\mathbf{R}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ and $\mathbf{T}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ from $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}$ and obtains the translation difference $D(i)$ and the rotation difference $A(i)$, respectively as

$$D(i) = \sum_{\substack{j=1 \\ j \neq i}}^{N} \text{dist}\left(\mathbf{T}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}, \mathbf{T}^{\mathcal{M}}_{\mathcal{I}(t_1|t_j)}\right) \tag{7}$$

$$A(i) = \sum_{\substack{j=1 \\ j \neq i}}^{N} \text{angle}\left(\mathbf{R}^{\mathcal{M}}_{\mathcal{I}(t_1|t_i)}, \mathbf{R}^{\mathcal{M}}_{\mathcal{I}(t_1|t_j)}\right) \tag{8}$$

The client ranks $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_i)}$ with $D(i)$ and $A(i)$, resulting in r_i^{trans} and r_i^{rot}, respectively. Finally, it selects the best result $\mathbf{P}^{\mathcal{M}}_{\mathcal{I}(t_c)}$ where $c = \arg\min_i(r_i^{\text{trans}} + r_i^{\text{rot}})$, and converts it to the calibration offset $\mathbf{P}^{\mathcal{E}}_{\mathcal{I}(t_c)}$ by using Equation 3.

5 SYSTEM IMPLEMENTATION

Most smartphone operating systems today (e.g., Android) already perform data fusion and provide virtual sensors. We implement MARVEL in Android, and use the provided rotation vector sensor for absolute rotation in the earth frame and the linear acceleration sensor for acceleration without gravity. Our app reads them as fast as possible, at a rate of 200 samples per second on a Lenovo Phab 2 Pro. We use OpenCV manager (with OpenCV 3.2.0), an Android application that provides a system-wide service to perform OpenCV computation for all applications. The optical flow uses the Lucas-Kanade algorithm [32] provided by OpenCV. All processed images are at a resolution of 640×360. Note that low-resolution images are sufficient for accurate image localization [15], and MARVEL still displays identified labels on a high-resolution camera feed (e.g., 1080×1920 on Lenovo Phab 2 Pro). To ensure fast image processing, we get raw pixels from the camera hardware (e.g., without JPEG

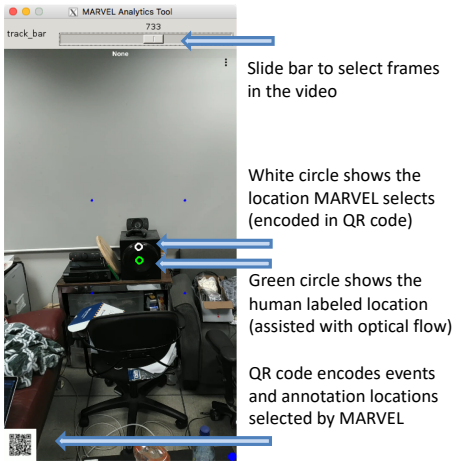


Figure 6: Evaluation tool to browse images, display identified labels, and manually label ground truth locations, assisted by optical flow.

compression), and compress only the images we decide to offload using OpenCV. Even though the Lenovo Phab 2 pro offers a depth camera, we do not use it in MARVEL.

On the server side, we build on top of SnapLink [15], an open source image localization system designed for appliance control. It provides a GRPC API, and our Android client uses GRPC 1.3.0 to communicate with it. The server is implemented in C++11 and uses OpenCV 3.3.0-rc and PCL 1.8.0.

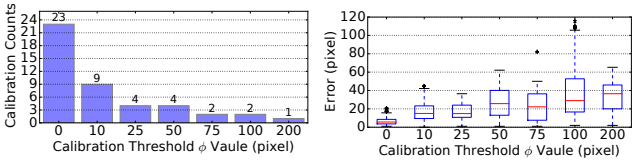
6 EVALUATION

In this section, we conduct several micro-benchmarks to demonstrate the effectiveness of the optimizations proposed in this paper, as well as end-to-end performance evaluations of MARVEL.

6.1 Experimental Tool and Setup

To ensure accurate annotation placement in real time, our localization system must achieve both high accuracy and low latency. Even with an accurate localization result, high latency causes an annotation view to be outdated, making it inaccurate from the user's perspective.

To measure the placement error, we capture a video of the smartphone screen while using MARVEL and analyze the video using a tool we built, as shown in Figure 6. The tool reads all frames from the video and shows a slide bar for frame selection. In each frame, there is a QR code on the left bottom corner, which has MARVEL-relevant information (e.g., annotation location on the screen, offloading event, and ZUPT event). The tool decodes the QR code, extracts the annotation locations given by MARVEL, and displays them on the screen, shown as the white circle in Figure 6. To get the ground truth, we manually click on the correct annotation location on a frame, shown as a green circle, and use optical flow to automatically mark the ground truth locations in all of the following frames. When optical flow fails at a frame due to a blurred image, we manually mark the correct location in the frame and restart optical flow-based labeling. We perform this labeling procedure until the ground truth locations in all frames are labeled correctly. Then, the placement error is the distance (in pixels) between the green circle (ground truth) and the white circle (MARVEL output)



(a) Number of Calibrations (b) Label Placement Error

Figure 7: Number of calibrations happened during a 20-second linear movement, and label placement errors with different calibration threshold ϕ. We select $\phi = 25$ for its low calibration overhead and low errors.

in a frame. All images have a resolution of 640×360, the same as in Section 2.3.

We deploy the server in the same local network on a Ubuntu machine with an Intel Xeon E5-2670 CPU and 256 GB of memory. It runs inside a Docker container (version number 17.09.0-ce). The smartphone communicates with the server using GRPC. When deploying the server, we collect a 3D model in a room of a campus building (Soda Hall at UC Berkeley). Given that MARVEL's performance depends on 6-DOF localization of the mobile device, rather than the number of target objects, we label only one object (a speaker) in the room for our micro-benchmarks, without loss of generality.

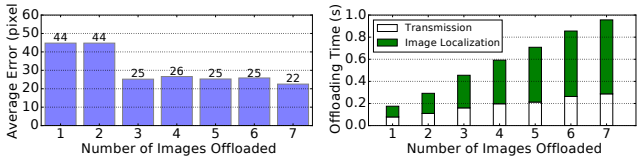
6.2 Calibration Threshold ϕ

Given that image offloading is necessary for calibration but consumes nontrivial energy, we study how the calibration threshold ϕ (in pixels) in Section 4.5 impacts the calibration performance. To show the effect of calibration, *we assume that image localization is accurate but inertial localization is erroneous.* To this end, in this experiment, we use an AprilTag [38] (not the speaker in Figure 6) as the target object to minimize image localization errors. We set $N = 1$, the minimum N value, since it is enough to provide accurate image localization with the AprilTag. We perform the same linear movement for 20 seconds, with seven different ϕ values.

Figures 7(a) and 7(b) show the number of calibrations and the annotation placement errors according to the threshold ϕ, respectively. As expected, with a lower ϕ value, MARVEL client becomes more sensitive to the difference between optical flow and inertial tracking results. Therefore, as the ϕ value increases, fewer calibrations are triggered, and larger errors are observed. In addition, when the ϕ value increases up to 25 pixels, the number of calibrations is reduced more than 80% (from 23 to 4) without a significant loss of accuracy. Based on the results, we choose 25 pixels as the default calibration threshold in MARVEL for the later experiments. Note that this threshold value is small enough for accurate annotation placement since 25×25 pixels is only 0.27% of a 640×360 image.

6.3 Number of Images for Localization

Next, we remove the assumption of ideal image localization and consider annotation placement for a real target object (speaker). As discussed in Section 4.5, the MARVEL client offloads N selected images for calibration to improve image localization accuracy by cross-validating image localization results. However, there is a trade-off between accuracy and latency when deciding N. As N



(a) Average Error (pixel) (b) Offloading Time Breakdown

Figure 8: Error and offloading time with different number of offloaded images. Transmitting 3 images yields good accuracy as well as reasonable offloading time.

increases, accuracy may increase but image uploading time (communication overhead) and cloud processing time may also increase. To find the optimal number of images to offload, we conduct a micro-benchmark by uploading the top 7 images among the recent 20 images, and obtain all the returned 7 image locations. To get the performance of offloading N (≤ 7) images, we use the returned locations of only the top N images to compute the label location and encode it into the QR code. Therefore, the QR code contains 7 label locations at every moment.

Figure 8(a) shows the average placement errors of 109 samples with different values for N. Offloading 1 or 2 images yields larger errors on image localization than offloading more than 3 images. This is because when selecting offloading images, MARVEL does not directly extract feature uniqueness but uses an indirect method (i.e., gyroscope and edge detection) to avoid heavy local image computation. The 1 or 2 images chosen by using the indirect method may contain insufficient unique features, resulting in inaccurate image localization. Offloading more images makes it more likely that one of them will be easy to localize on the server, reducing localization error. In our experiment, offloading 3 images is good enough for accurate calibration even with these indirect methods.

Figure 8(b) shows the offloading time according to N. It shows that transmitting more images incurs higher offloading time, which can be problematic not only because it causes more energy consumption for communication but also because the returned calibration result can be too stale with accumulated IMU errors. Empirically, we observe acceptable IMU errors within 500 ms. Given that offloading time is just below 500 ms when transmitting 3 images, we set $N = 3$ for the later experiments. On the other hand, this verifies that MAR's latency should be decoupled from offloading latency, unlike in previous work [16, 18, 26, 49]. Note that in MARVEL, this 500 ms latency does not affect the system's latency (from the user's perspective), which is evaluated in Section 6.4.

6.4 End-to-End Latency

One of our design goals is to minimize the *end-to-end* localization latency, measured from the time of the sensor sampling (i.e., IMU or camera) to the time when identified annotations are displayed on the smartphone screen. In practice, the end-to-end latency includes not only the localization process but also *the underlying operating system behaviors*, such as task scheduling and memory copy. The underlying operation can be categorized into two parts: (1) moving sampled data (e.g., IMU data or image) from sensor buffer to user space memory, and (2) moving images from user space memory to screen. These factors were overlooked in previous work [16, 26].

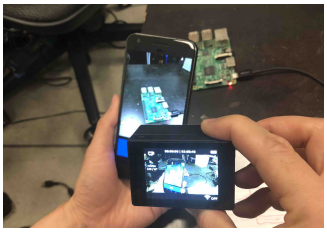


Figure 9: We record a 240 FPS video to measure the time for a basic camera app to reflect an LED change.

		Pixel (ms)	Phab 2 Pro (ms)	
	Operation	Wi-Fi	Wi-Fi	Cellular
Baseline	Camera->App	67.3	71.0	
	App->Cloud->App	177.7	287.8	4299.5
	App->Screen	36.9	70.7	
	Total	**281.9**	**429.5**	**4441.2**
MARVEL	IMU->App	8.7	12.6	
	IMU Computation	0.3	0.4	
	App->Screen	36.9	70.7	
	Total	**45.9**	**83.7**	

Table 3: Average latency in millisecond at different steps in the baseline system and MARVEL. MARVEL has lower latency because it performs identification using only local information, including the calibration offset $\mathbf{P}^{\mathcal{C}}_{\mathcal{I}(t_0)}$.

To measure the time of (1), denoted as $t_{\text{IMU->App}}$ and $t_{\text{Camera->App}}$, we compute

$$t_{\text{sensor->App}} = t_{\text{app\_callback}} - t_{\text{sensor\_event}} \quad (9)$$

where 'sensor' can be either 'IMU' or 'Camera'. $t_{\text{sensor\_event}}$ is the timestamp of sampling recorded in the hardware driver in Android and $t_{\text{app\_callback}}$ is acquired at the invocation of sensor data callback function. The results are shown in Table 3, for Google Pixel and Lenovo Phab 2 Pro.

To measure the time of (2), denoted as $t_{\text{App->Screen}}$, we measure '$t_{\text{Camera->App}} + t_{\text{App->Screen}}$' using a basic camera application,[4] which continuously reads images from the camera to memory and sends them to the screen. Specifically, as shown in Figure 9, a smartphone points at a Raspberry Pi which is blinking its red LED. To measure the delay between an LED blinking event and its display on the smartphone screen, a GoPro Hero 3+ high frequency camera (240 FPS) monitors both the smartphone screen and the LED. We count how many (GoPro) frames are taken for the smartphone screen to reflect an LED change. We observe that Google Pixel and Lenovo Phab 2 Pro require 104.2 ms and 141.7 ms of average delay, respectively. We then subtract the camera latency $t_{\text{Camera->App}}$ from the total time to get $t_{\text{App->Screen}}$, resulting in 36.9 ms and 70.7 ms for Google Pixel and Lenovo Phab 2 Pro, respectively. It is impossible to accurately measure the time to update a UI component (e.g., drawing a box) from user space memory to screen, which we assume is not longer than (2) because image data is usually larger.

Table 3 shows the end-to-end latency of a baseline MAR and MARVEL running on two different smartphones. The baseline MAR relies highly on the server, making the smartphone simply offload

[4] https://github.com/googlesamples/android-Camera2Basic

an image to a local server and wait until getting the image localization and object identification result, which is the same as previous MAR work [15, 26, 27]. We measure the time delay from serializing an image localization request to the return of identified labels, denoted as $t_{\text{App->Cloud->App}}$. Table 3 shows that $t_{\text{App->Cloud->App}}$ of the baseline system is significantly larger and highly variable with different smartphone models and Internet connectivity. This degrades end-to-end latency performance, resulting in 281.9 ms to even 4441.2 ms, far above 100 ms. This confirms that relying on the cloud is not a good choice for real-time MAR.

On the other hand, MARVEL's latency does not depend on how long a calibration (cloud offloading) takes, because the smartphone always has the calibration offset $\mathbf{P}^{\mathcal{E}}_{I(t_0)}$ available, as discussed in Section 4.2. The only additional latency besides memory copy comes from the integration of sensor readings, which takes no more than 0.4 ms on both smartphones. As a result, MARVEL provides 45.9 ms and 83.7 ms latency for each smartphone model, which are lower than our goal of 100 ms. Note that this latency is even lower than the basic camera app, which means that the camera app's latency (which we cannot control) will be the overall latency of MARVEL.

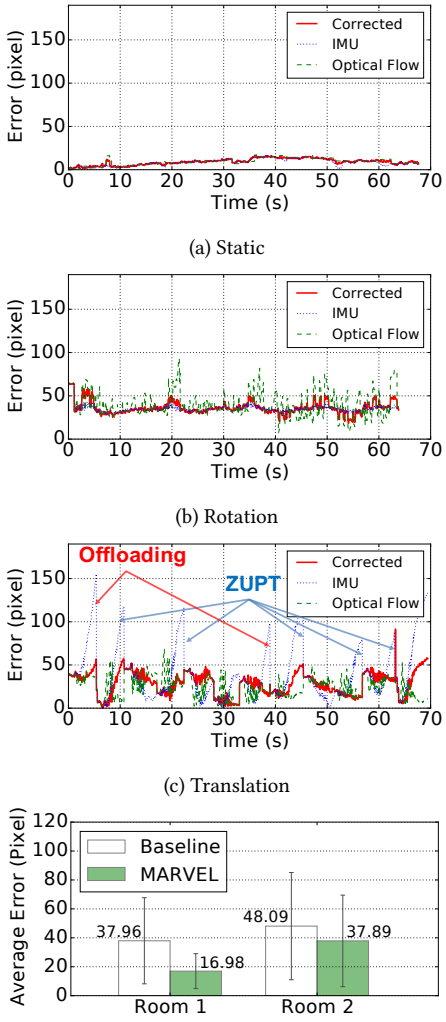
6.5 Annotation Placement Accuracy

In this Section, we evaluate MARVEL effectiveness, in terms of annotation placement accuracy, in various scenarios. Note that the placement accuracy incorporates both localization accuracy and end-to-end latency. We use $\phi = 25$ and $N = 3$ for MARVEL.

Various Movement Patterns. We conduct three micro-benchmarks with different device movement patterns: (1) holding still, (2) rotating only at a natural speed, and (3) moving around (i.e., translate) at a natural speed. Figures 10(a)-(c) show the placement errors over a 60 second period of the experiment after app initialization for the three different operations. For comparison, we plot the errors for three types of results, 'IMU' ($\mathbf{s}^L_{\text{imu}}(t)$), 'Optical Flow' ($\mathbf{s}^L_{\text{of}}(t)$), and 'Corrected' ($\mathbf{s}^L_{\text{final}}(t)$). To clearly visualize the effectiveness of our label placement correction using IMU and optical flow, we use calibration threshold $\phi = 100$ to induce less image offloading.

Figure 10(a) shows that MARVEL maintains almost perfect accuracy when the device does not move. Figure 10(b) shows that when the device rotates, the error increases compared to the static case but is well bounded for a long time. No ZUPT or image offloading event occurs. This verifies that the IMU's rotation data (from fusing gyroscope and gravity) is accurate for a long time. In addition, Figure 10(b) shows that 'Optical Flow' generates larger errors than 'IMU'. Since optical flow fails with large image changes or blurred images, the accuracy of 'Optical Flow' is degraded when the mobile device rotates quickly. Note that rotating a mobile device changes images on the screen faster than linearly moving it. As expected, 'Corrected' generates a better performance than 'Optical Flow' when the device is rotating. Since 'Corrected' combines IMU and optical flow results by using their own confidence values, inertial tracking compensates optical flow's errors with a higher confidence value.

Figure 10(c) shows that when moving the mobile device with translation, the situation becomes different. Now 'IMU' accumulates errors (coming from cumulative errors of the accelerometer) while 'Optical Flow' works better than in Figure 10(b). It also shows



(a) Static

(b) Rotation

(c) Translation

(d) Error Comparison

Figure 10: (a)-(c) show annotation placement error (pixel) while conducting different actions, and (d) shows error comparison between a baseline system and MARVEL.

that 'IMU' errors increase quickly but are eventually corrected by either image offloading (1st and 4th spikes of IMU errors) or ZUPT (other spikes of IMU errors). The local ZUPT events often occur before MARVEL detects visual-inertial inconsistency, avoiding unnecessary offloading events. Compared to previous MAR systems relying on cloud offloading for every object recognition operation [16, 18, 26, 49], MARVEL's selective offloading can reduce communication overhead significantly. When 'IMU' accumulates errors, 'Corrected' still maintains good accuracy by combining optical flow-based results with higher confidence values. There are also some points where 'IMU' is better than 'Optical Flow', where 'Corrected' is not affected by optical flow's errors. Overall, these results verify that our idea of using inertial and visual tracking together for real-time annotation placement is valid in various movement scenarios.

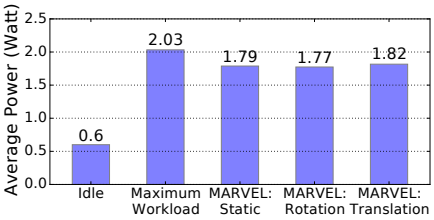


Figure 11: MARVEL incurs less optical flow and offloading than maximum workload (continuously optical flow and offloading).

Comprehensive Evaluation. We also conduct experiments in a more comprehensive setting: two target objects (i.e., a speaker and a fan) in two different rooms, with natural motions including static periods, rotations, and translations. Note that each room provides its own 3D point cloud database, which impacts image localization performance. As in Section 6.4, we add the baseline system for comparison, which continuously offloads one image to a local server through WiFi and displays recognized annotations when the server returns a result (without local optical flow).

Figure 10(d) shows the mean and standard deviation of the placement errors when using the baseline and MARVEL, respectively. It shows that the accuracy becomes different when the room changes, which confirms the impact of a background image on the image localization. On the other hand, MARVEL always outperforms the baseline system. Although MARVEL incurs less image localization than the baseline, it provides higher accuracy due to the help of fast IMU processing and selective optical flow. Note that the baseline performance can be even worsened when the server is not deployed locally, since higher network latency degrades placement accuracy. This verifies that local computing is necessary to achieve accurate results in time-sensitive MAR applications.

6.6 Power Consumption

We measure MARVEL's power usage while performing different actions. As in Section 2.3, we use the Trepn Profiler to measure the system-wide power consumption on a Lenovo Phab 2 Pro Android phone. To ensure consistent power consumption by the display [12], we set the screen brightness to the same value, and conduct all energy measurements in the same room with all lights turned on.

We perform the following five actions: (1) idle (only system background tasks with the screen turned on, for comparison), (2) maximum workload (continuously performing optical flow and image offloading), (3) static (pointing the smartphone to the object without moving, i.e., only inertial tracking), (4) rotation at a natural speed (inertial tracking, with occasional optical flow and image offloading), and (5) moving (i.e., translate) at a natural speed (inertial tracking, with occasional optical flow and image offloading). Each action is performed for around 90 seconds.

Figure 11 shows the average power usage of these actions. As expected, the maximum workload consumes significant power, 2.03 Watts. MARVEL, regardless of movement patterns, consumes around 1.79 Watts on average by mainly using IMU rather than optical flow and image offloading, which is a 11.8% improvement. Due to our selective optical flow and image offloading mechanisms, rotation and translation do not add significant power consumption over the static scenario.[5] In comparison, when operating with Wi-Fi connection, Overlay [26] consumes 4.5 Watt, and Glimpse [16] consumes 2.1 Watt.

7 RELATED WORK

As a *holistic* MAR system, MARVEL is built on, but clearly differentiated from, a large body of prior work including computer vision, cloud offloading, and indoor localization.

Computer Vision: Computer vision technology is a key enabler to identify an object on the screen. There are three common methods of object identification: image retrieval, CNN, and image localization. To the best of our knowledge, prior MAR work exploits image retrieval (e.g., Overlay [26], VisualPrint [27], and CloudAR [49]) or CNN (e.g., Glimpse [16]). However, as discussed in [15], image retrieval and CNNs require a much larger database than image localization to achieve accurate instance identification. Furthermore, when these two approaches are applied to MAR, both database and computation are offloaded to the cloud due to the limited capability of mobile devices, making an MAR system unable to identify objects without using the cloud, generating long identification latency.

In contrast, since image localization identifies an object using its *location*, it can be easily combined with other localization methods constructively. MARVEL takes this advantage and combines it with IMU-based localization, which can be quickly performed *locally*. This design choice enables the object identification procedure to be much less dependent on the cloud.

Computation and Storage Offloading: A common way to overcome the computation and storage limitations on smartphones is offloading, either to the cloud or edge devices (e.g., in the same network). While cloud offloading has been common in the MAR regime due to the limited capabilities of mobile devices [25], it induces significant latency and energy consumption for communication [9, 36]. A number of studies have tried to use local computing to alleviate the problem. Overlay [26] obtains the mobile device's location from its sensor data, using it to offloading fewer images and reducing the visual search space in the cloud. VisualPrint [27] locally processes an image to extract its most distinct visual features and offloads these features instead of the complete image, which shifts offloading overhead to local image computation overhead. Glimpse [16] and CloudAR [49] rely on the cloud to identify an object but locally keep track of the identified object's location on the screen by using optical flow, enabling the object's annotation to move accordingly in real time, regardless of offloading latency. However, all of them still suffer from long latency (dependent on offloading) for identifying a new object on the screen. In all prior work, the cloud takes *the main role* to identify objects and local computing performs *auxiliary work* to help the cloud; these systems cannot identify an object without using the cloud.

More generic cloud offloading work tries to find a sweet spot between offloading overhead and local computation overhead by offloading opportunistically according to network condition; offloading when network condition is favorable to make it more efficient than local computing [17, 19]. These adaptive techniques

[5] We believe that rotation consumes 0.02 Watt less power than static simply due to power measurement noise.

cannot be applied to the above MAR work (image retrieval- or CNN-based identification) where using the cloud is always required. It also cannot be used for MARVEL which triggers offloading when calibration is needed, regardless of network conditions.

Other MAR works have explored computation and storage offloading to nearby *edge* devices rather than the cloud, which runs without the Internet. For example, in MARBLE [41], local wall-powered BLE beacons are used as edge devices, compressing and storing 3D representations of objects. A smartphone receives the compressed 3D representations from nearby BLE beacons, decompresses them, and overlays them on the screen. Although this 3D rendering application is much more demanding than an annotation service and MARBLE does not reduce a smartphone's burden sufficiently, the idea of utilizing edge devices to be independent from the Internet is generally applicable in MAR.

In contrast to the prior work, MARVEL lets local computing act as the main role for object identification (inertial localization) with the cloud assisting it (sporadic calibration based on image localization). This approach decouples identification latency from offloading latency and significantly reduces offloading overhead.

Indoor Localization: Indoor localization is a very well-researched topic. Along with various other techniques, using IMUs on smartphones has also been investigated [31, 42, 43, 47]. While rotation is quite accurate by fusing gyroscope and gravity, a linear accelerometer is still error-prone due to cumulative errors [43], which makes inertial sensor-based localization valid only for a short period of time [31]. Various methods have been explored to calibrate the IMU errors, such as Wi-Fi signal strength [42] and light sensors [47]. Nevertheless, none of them achieve sufficient accuracy to support MAR. In contrast, MARVEL's image-based calibration significantly improves the accuracy of inertial localization.

On the other hand, the robotics community has shown that Visual-Inertial SLAM (VISLAM) can achieve accurate and real-time indoor navigation locally on relatively powerful computers [7, 18, 30]. State-of-the-art VISLAM uses both visual and inertial localization and fuses them together using a Bayes Filter (e.g., the Extended Kalman Filter [33]). However, these techniques are too heavyweight to operate on ordinary mobile devices due to their limited storage and computation capabilities.

Powerful CPUs and GPUs in the latest smartphones allow MAR systems, such as Google ARCore [2] and Apple ARKit [4], to perform VISLAM locally. Unlike MARVEL, they focus on rendering 3D objects in a small area and do not need cloud storage. In fact, they use the cloud to share coordination systems among users for in-app cooperation, such as games [3, 5]. To push this idea further, companies and organizations have started to explore how to store 3D models in the cloud and share them among MAR applications, such as AR Cloud [1, 39]. This will make MARVEL even easier to deploy with 3D models collected and maintained by third parties.

8 DISCUSSION AND FUTURE WORK

In this section, we discuss some limitations of MARVEL, which inspire directions for future work.

Image Localization Errors: Image localization has improved significantly with the advances in computer vision and robotics. However, the accuracy of an image localization operation still depends on many factors. For example, not only the query image, but the environment itself must contain enough unique features [27]. The 3D model constructed in advance must cover the features generated from different viewing angles. Errors caused by those factors are not always easy to eliminate. Therefore, we must be able to detect those errors and react in a fast and efficient way.

Integration with Other Information: While performing image localization, extra information can be used to verify the result and detect errors. For example, visual markers (e.g., AprilTag [38]) are a robust and efficient way to provide another source of 6-DOF location, but their deployment can be intrusive. CNN-based category recognition is useful to verify or correct 2D label placement, but can only be done efficiently in the cloud. Optical Character Recognition [35] can extract from images, which can also be very helpful when unique text appears in a query image.

Moved Objects: Localization-based instance recognition has a strong assumption that objects do not move. We recognize this as a limitation, but argue that many objects in our usage scenario are not usually moved. For example, exhibited items in a museum do not change often. Large appliances in a building, such as projectors and printers, also do not move frequently. Even if they are moved, obtaining an updated 3D model of a room only involves capturing a short video as well as clicking and typing several annotations [15]. Integrating MARVEL with neural network-based systems, such as Glimpse [16], can also help to recognize moving objects.

9 CONCLUSION

In this paper, we presented MARVEL, an MAR system for a real-time annotation service, which is designed to run on regular mobile devices while achieving low latency and low energy consumption. Unlike previous MAR systems which suffer from high offloading latency and large energy consumption for image offloading and local image computation, MARVEL mainly uses local inertial data processing which is much faster (from 250 ms to <100 ms) and consumes less energy. While continuously and quickly placing annotations via local inertial tracking, MARVEL compensates for its errors by using optical flow and image offloading. However, in doing so, MARVEL does its best to avoid any redundant image computation and image offloading. In MARVEL, a mobile device maximizes the offloading interval, selects only a few query images when offloading, avoids image computation when selecting these query images, and minimizes optical flow computations. Experimental results show that the MARVEL system design enables low energy consumption without sacrificing accuracy.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers and our shepherd, Prof. Karthik Dantu, for their constructive comments and help on finalizing this paper. We are also thankful to John Kolb and Sam Kumar for proofreading this paper and providing feedback. This work is supported in part by the National Science Foundation under grant CPS-1239552 (SDB) and California Energy Commission, and in part by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education (NRF-2016R1A6A3A03007799).

REFERENCES

[1] 6dai 2018. 6D.ai. https://www.6d.ai. (2018).
[2] arcore 2017. Google ARCore. https://developers.google.com/ar/. (2017).
[3] arcorecloudanchor 2018. Google ARCore Cloud Anchors. https://codelabs.developers.google.com/codelabs/arcore-cloud-anchors/index.html. (2018).
[4] arkit 2017. Apple ARKit. https://developer.apple.com/arkit/. (2017).
[5] arkit2 2018. Apple ARKit 2. https://www.apple.com/newsroom/2018/06/apple-unveils-arkit-2/. (2018).
[6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. *Computer vision–ECCV 2006* (2006), 404–417.
[7] Gabriele Bleser. 2009. *Towards visual-inertial slam for mobile augmented reality*. Verlag Dr. Hut.
[8] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
[9] Tristan Camille Braud, Dimitrios Chatzopoulos, Pan Hui, et al. 2017. Future Networking Challenges: the Case of Mobile Augmented Reality. In *Proceedings-International Conference on Distributed Computing Systems*.
[10] Stuart K Card. 2017. *The psychology of human-computer interaction*. CRC Press.
[11] Stuart K Card, George G Robertson, and Jock D Mackinlay. 1991. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*. ACM, 181–186.
[12] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855840.1855861
[13] Aaron Carroll and Gernot Heiser. 2013. The systems hacker's guide to the galaxy energy usage in a modern smartphone. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 5.
[14] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. 2017. Mobile Augmented Reality Survey: From Where We Are to Where We Go. *IEEE Access* (2017).
[15] Kaifei Chen, Jonathan Fürst, John Kolb, Hyung-Sin Kim, Xin Jin, David E Culler, and Randy H Katz. 2017. SnapLink: Fast and Accurate Vision-Based Appliance Control in Large Commercial Buildings. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 4 (2017), 129:1–129:27.
[16] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 155–168.
[17] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 301–314.
[18] Alejo Concha, Giuseppe Loianno, Vijay Kumar, and Javier Civera. 2016. Visual-inertial direct SLAM. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 1331–1338.
[19] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 49–62.
[20] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 123–136.
[21] Richard Hartley and Andrew Zisserman. 2003. *Multiple view geometry in computer vision*. Cambridge university press.
[22] hololens 2017. Microsoft HoloLens. https://www.microsoft.com/en-us/hololens. (2017).
[23] Berthold KP Horn and Brian G Schunck. 1981. Determining optical flow. *Artificial intelligence* 17, 1-3 (1981), 185–203.
[24] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. 2015. The case for offload shaping. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 51–56.
[25] Zhanpeng Huang, Weikai Li, Pan Hui, and Christoph Peylo. 2014. CloudRidAR: A cloud-based architecture for mobile augmented reality. In *Proceedings of the 2014 workshop on Mobile augmented reality and robotic technology-based systems*. ACM, 29–34.
[26] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2015. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 331–344.
[27] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2016. Low Bandwidth Offload for Mobile AR. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 237–251.
[28] Karthik Kumar, Yamini Nimmagadda, and Yung-Hsiang Lu. 2012. Energy conservation for image retrieval on mobile systems. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 3 (2012), 66.
[29] Mathieu Labbe and Francois Michaud. 2013. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics* 29, 3 (2013), 734–745.
[30] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. 2015. Keyframe-based visual–inertial odometry using nonlinear optimization. *The International Journal of Robotics Research* 34, 3 (2015), 314–334.
[31] Fan Li, Chunshui Zhao, Guanzhong Ding, Jian Gong, Chenxing Liu, and Feng Zhao. 2012. A reliable and accurate indoor localization method using phone inertial sensors. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 421–430.
[32] Bruce D Lucas, Takeo Kanade, et al. 1981. An iterative image registration technique with an application to stereo vision. In *Proceedings DARPA image understanding workshop*. Vancouver, BC, Canada, 121–130.
[33] João Luís Marins, Xiaoping Yun, Eric R Bachmann, Robert B McGhee, and Michael J Zyda. 2001. An extended Kalman filter for quaternion-based orientation estimation using MARG sensors. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, Vol. 4. IEEE, 2003–2011.
[34] Robert B Miller. 1968. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 267–277.
[35] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. 1999. *Optical character recognition*. John Wiley & Sons, Inc.
[36] Nayyab Zia Naqvi, Karel Moens, Arun Ramakrishnan, Davy Preuveneers, Danny Hughes, and Yolande Berbers. 2015. To cloud or not to cloud: a context-aware deployment perspective of augmented reality mobile applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 555–562.
[37] Jakob Nielsen. 1994. *Usability engineering*. Elsevier.
[38] Edwin Olson. 2011. AprilTag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 3400–3407.
[39] openarcloud 2018. Open AR-Cloud. https://open-arcloud.org/. (2018).
[40] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. 2011. Fast image-based localization using direct 2d-to-3d matching. In *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 667–674.
[41] Chong Shao, Bashima Islam, and Shahriar Nirjon. 2018. MARBLE: Mobile Augmented Reality Using a Distributed BLE Beacon Infrastructure. In *Internet-of-Things Design and Implementation (IoTDI), 2018 IEEE/ACM Third International Conference on*. IEEE, 60–71.
[42] Guobin Shen, Zhuo Chen, Peichao Zhang, Thomas Moscibroda, and Yongguang Zhang. 2013. Walkie-markie: indoor pathway mapping made easy. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 85–98.
[43] Isaac Skog, Peter Handel, John-Olof Nilsson, and Jouni Rantakokko. 2010. Zero-velocity detection - An algorithm evaluation. *IEEE Transactions on Biomedical Engineering* 57, 11 (2010), 2657–2666.
[44] Irwin Sobel and Gary Feldman. 1968. A 3x3 isotropic gradient operator for image processing. *a talk at the Stanford Artificial Project in* (1968), 271–272.
[45] tango 2017. Google Project Tango. https://get.google.com/tango/. (2017).
[46] trepn 2018. Qualcomm Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler. (2018).
[47] Qiang Xu, Rong Zheng, and Steve Hranilovic. 2015. IDyLL: Indoor localization using inertial and light sensors on smartphones. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 307–318.
[48] Wenxiao Zhang, Bo Han, and Pan Hui. 2017. On the Networking Challenges of Mobile Augmented Reality. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*. ACM, 24–29.
[49] Wenxiao Zhang, Sikun Lin, Farshid Hassani Bijarbooneh, Hao Fei Cheng, and Pan Hui. 2017. CloudAR: A Cloud-based Framework for Mobile Augmented Reality. In *Proceedings of the on Thematic Workshops of ACM Multimedia 2017 (Thematic Workshops '17)*. ACM, New York, NY, USA, 194–200. https://doi.org/10.1145/3126686.3126739
[50] Liang Zheng, Yi Yang, and Qi Tian. 2017. SIFT meets CNN: A decade survey of instance retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).