MIT Libraries | DSpace@MIT

MIT Open Access Articles

Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Chen, Tiffany Yu-Han, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. "Glimpse." Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems -SenSys '15 (2015).

As Published: http://dx.doi.org/10.1145/2809695.2809711

Publisher: Association for Computing Machinery

Persistent URL: http://hdl.handle.net/1721.1/110758

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices

Tiffany Yu-Han Chen
MIT CSAIL
yuhan@csail.mit.edu

Lenin Ravindranath
Microsoft Research
lenin@microsoft.com

Shuo Deng
MIT CSAIL
shuodeng@csail.mit.edu

Paramvir Bahl
Microsoft Research
bahl@microsoft.com

Hari Balakrishnan
MIT CSAIL
hari@csail.mit.edu

ABSTRACT

Glimpse is a continuous, real-time object recognition system for camera-equipped mobile devices. Glimpse captures full-motion video, locates objects of interest, recognizes and labels them, and tracks them from frame to frame for the user. Because the algorithms for object recognition entail significant computation, Glimpse runs them on server machines. When the latency between the server and mobile device is higher than a frame-time, this approach lowers object-recognition accuracy. To regain accuracy, Glimpse uses an *active cache* of video frames on the mobile device. A subset of the frames in the active cache are used to track objects on the mobile, using (stale) hints about objects that arrive from the server from time to time. To reduce network bandwidth usage, Glimpse computes *trigger frames* to send to the server for recognizing and labeling. Experiments with Android smartphones and Google Glass over Verizon, AT&T, and a campus Wi-Fi network show that with hardware face detection support (available on many mobile devices), Glimpse achieves precision between 96.4% to 99.8% for continuous face recognition, which improves over a scheme performing hardware face detection and server-side recognition without Glimpse's techniques by between 1.8-2.5×. The improvement in precision for face recognition without hardware detection is between 1.6-5.5×. For road sign recognition, which does not have a hardware detector, Glimpse achieves precision between 75% and 80%; without Glimpse, continuous detection is non-functional (0.2%-1.9% precision).

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems; Interactive systems*

Keywords

mobile computing; wearable computing; cloud computing; caching; Google Glass

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SenSys'15, November 1–4, 2015, Seoul, South Korea.
© 2015 ACM. ISBN 978-1-4503-3631-4/15/11 $15.00
DOI: http://dx.doi.org/10.1145/2809695.2809711.

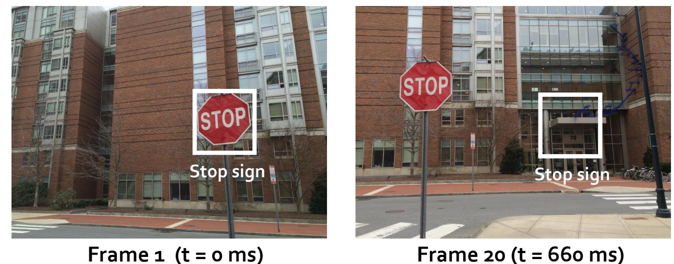


Figure 1: Offloading every frame to a server reduces trackability (right): the stop sign's location is wrong.

1. INTRODUCTION

Cameras of good quality are now available on almost every handheld and wearable mobile device. The high resolution of these cameras coupled with pervasive wireless connectivity makes it feasible to develop continuous, real-time object recognition applications. These applications *locate* objects in a video stream and *label* them with information associated with the objects. For example, an application that helps a user assemble furniture can pinpoint and label each piece [53, 6], a driver assistance application that locates and labels road signs can improve driver safety [28], a tourist application that recognizes landmarks and buildings can improve user experience, and so on. Researchers have also proposed applications of object recognition for smart homes [45, 42] and perceptual user interfaces [54, 36].

To support these applications, the object recognition system must provide high *trackability*, i.e., it should be able to locate an object accurately and label it with an identifier. Achieving high trackability in real-time on mobile devices is challenging because the required computer-vision algorithms are computationally intensive and must run at the real-time rate of 30 frames per second (see §2.3). Hardware support for face detection (locating a face) is available today, but labeling a face scales with the size of the corpus of faces, and is infeasible on a mobile device.

For these reasons, object recognition tasks must be offloaded to more powerful servers. Offloading is a well-known idea [13, 23, 22], but in the context of continuous recognition, it must be applied with care because wireless network latencies are too high. For example, if it takes 700 milliseconds to transfer a frame and recognize its objects at a server (measured time on an LTE network), when the results arrive they will be over 20 frames old, and the located object may

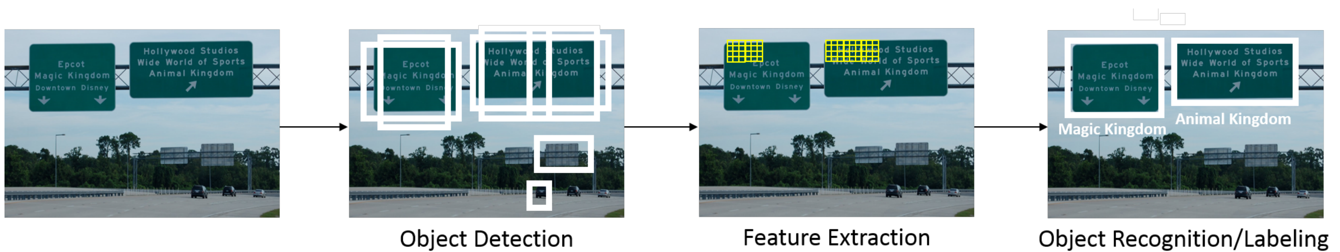


Figure 2: The three computationally-intensive stages of the object recognition pipeline.

no longer be at the reported position. An example is shown in Figure 1: the user sees incorrect results.

This paper presents the design, implementation, and evaluation of Glimpse, a continuous, real-time object recognition system that achieves high trackability, while reducing the amount of network bandwidth consumed relative to alternative designs. Glimpse achieves high trackability by maintaining an *active cache* of frames on the mobile device, and computing over the cached frames using the stale hints that arrive from the server to get an estimate of an object's location in the current frame. Because tracking an object through every cached frame takes too long, the active cache subsamples frames using the rate of change between scenes, the network delay, and the device capability.

Glimpse reduces bandwidth consumption by strategically sending only certain *trigger frames* to the server to obtain object recognition hints. Trigger frames are ones for which the server's answer is likely to differ from the local tracking.

Glimpse occupies an interesting point in the design space of possible ways to split computation between the server and mobile. The server does the hard work of recognition and labeling, sending back both labeled objects and features to use in the mobile's active cache, but this information is always a little stale. The mobile tracks the objects locally using the active cache, not going through every cached frame, and whenever a response for a past frame arrives from the server, it "catches up" to the current frame and processes subsequent frames in real time.

The active cache and trigger frames are generic techniques that may be applied to a wide range of real-time, vision-based systems. Advances in computing hardware and vision algorithms are likely to change the placement of components in a system like Glimpse in the future. For instance, today, there is no hardware support on mobile devices for detection of objects other than faces, but that may change in the future. As long as the processing delay is significantly higher than the time between subsequent frames (33 ms), the results will be stale by the time it is shown to the user. The active cache will hide this latency from the user, while trigger frames reduce the number of frames that must be processed. We discuss the generality of these techniques in §9.

We have implemented Glimpse for recognizing road signs and faces and have conducted experiments with Android smartphones and Google Glass. We evaluate two versions of Glimpse, sw and hw. In both versions, object recognition (labeling) runs on the server. In the sw version, object detection for road signs and faces run on the server as well, whereas in the hw version (implemented only for faces), detection runs in mobile hardware.

Our main experimental results are:

1. Over campus Wi-Fi, Verizon LTE, and AT&T's LTE, Glimpse (hw) achieves precision between 96.4% to 99.8% for face recognition. By contrast, a scheme that performs hardware face detection and server-side recognition without Glimpse's techniques achieves precision between about 38.4% and 56.5%; the improvement with Glimpse (hw) is between 1.8-2.5×. These improvements are due to the active cache.
2. The improvement in precision for face recognition without hardware (sw) is between 1.6-5.5×. For road sign recognition, which does not have a hardware detector, the improvement is enormous: 42× for Wi-Fi and 375× for Verizon LTE (road sign recognition does not work in the baseline here). Again, the active cache is responsible for these improvements.
3. Trigger frames reduce the bandwidth by up to 2.3× and consumes 25-35% less energy compared to using the active cache alone, while retaining the same accuracy.

A video demonstration of Glimpse is at the URL, `http://people.csail.mit.edu/yuhan/glimpse`.

2. BACKGROUND AND DESIGN

This section describes the object recognition pipeline and tracking, and Glimpse's challenges and architecture.

2.1 Object recognition pipeline

The recognition pipeline for objects or faces, which runs on a video frame, consists of three stages: *detection*, *feature extraction*, and *recognition* (Figure 2).

Detection: This stage searches the frame for objects of interest and locates them with bounding boxes, but without labels. The object detector uses distinctive characteristics of objects such as a closed boundary for cars, a prominent color contrast from the background for road signs, and region differences for faces (the eyes are darker than the cheeks).

Feature extraction: This stage processes the content inside the bounding box computed by the detector to extract features that represent the object, using methods like the scale-invariant feature transform (SIFT) [34, 12] and speeded-up robust features (SURF) [7, 12].

Recognition/Labeling: This stage recognizes the object and assigns a label to it using a machine-learning classifier trained offline using a database of labeled object images. The training phase extracts feature vectors as mentioned above, after which it constructs a model such as a Support Vector Machine [18]. Online, it uses the trained model to convert feature vectors into a label.

2.2 Object tracking

The goal of object tracking is to follow a moving object from one frame to the next in a video stream. Glimpse

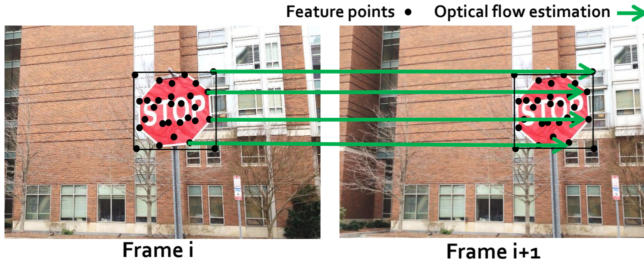


Figure 3: Extracting feature points on a road sign in frame i, computing the optical flow for each of the points, and locating the points in frame $i + 1$.

Scheme	Device-only	Offload to Server (Wi-Fi)
Road sign recognition	11.32 J	0.54 J
Face recognition	5.16 J	0.44 J

Table 1: Energy consumption of the object recognition pipeline for a single frame on a Samsung Galaxy Nexus.

uses the Lucas-Kanade tracking algorithm [35], which maps the location of the object from one frame to the next in two steps: (1) extract feature points representing the moving object [49], and (2) estimate where those feature points could be in the second frame to locate the object. For the first step, one can reuse the feature points and the bounding box obtained in the *feature extraction* and *detection* stages. Alternatively, one could apply algorithms such as *good features to track* [49], SIFT [34], or SURF [7] to locate corners and prominent points in the moving object. These features points are then tracked across frames using optical flow techniques [8, 24], which compute the velocity of points between frames.

The result of the tracking stage is a set of successfully tracked feature points in the second frame. Figure 3 shows an example of extracted feature points on a road sign, and the result of tracking. In general, object tracking is a faster operation compared to the recognition pipeline.

2.3 Challenges and approach

We ran the various recognition and tracking tasks described above on different platforms and measured their performance. Table 2 shows the results for Google Glass, a smartphone, and a server machine. We found that all stages showed significant processing time differences between the server and the mobile device. For example, running object detection on the device can be 11-21× slower than running it on a typical server machine; feature extraction can be 18× slower, and recognition can be 14× slower.

The increased processing time also leads to increased energy consumption. Table 1 compares the energy consumption for processing a single frame on the device and offloading. The energy consumed by executing the entire pipeline on the device is 12-21× more compared to energy consumed when each frame is offloaded to the server.

The feature extraction and the recognition stage have a large memory requirement. Beyond a point, the trained model will be too big to fit in a mobile device's memory. In addition, maintaining and updating the database of labeled objects is best done on the server.

Recently, some mobile device manufacturers have incorporated face detection in hardware [1]. The facility significantly reduces detection time by 6×, from 1129 ms to 175 ms (Table 2 rows 5 and 6). Feature extraction and recognition, however, are still expensive operations that are best executed on the server. Moreover, there are no hardware detectors for other objects other than faces on devices today.

Finally, we note that object tracking is fast on the mobile device (last row of Table 2).

These results demonstrate the importance of offloading object recognition tasks to the servers. Offloading, however, presents a key challenge: network delivery and server processing latency, which could be several hundred milliseconds (or more) for individual frames. Sending every frame to the server would significantly degrade trackability (Figure 1).

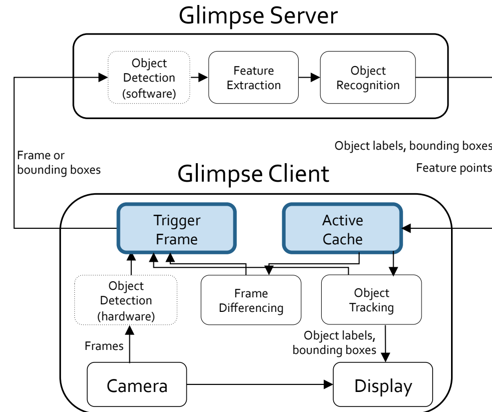


Figure 4: Glimpse Architecture. We explain active cache in §3 and trigger frame in §4.

Figure 4 shows the processing architecture of Glimpse. The Glimpse client runs on the mobile device, receives and stores frames captured by the device's camera, and sends *trigger frames* to the server. The server runs the stages of the recognition pipeline on each frame it receives, producing bounding boxes with labels as well as *feature points* for each recognized object. The client uses the feature points in its tracking phase through the *active cache*, processing only a carefully selected subset of frames to track, adjusting the bounding boxes to the current frame (§3), thereby hiding the network latency from the user. It continuously annotates the user display with the tracked bounding boxes and object labels.

The key aspects of Glimpse are its active cache and trigger frames, which are described in the next two sections.

3. ACTIVE CACHE

The active cache addresses the following problem: how to locate moving object(s) on the mobile device when it takes many frame-times to obtain information, which turns out to be stale, about the object(s) from the server?

Our approach tracks objects on the mobile device by computing the optical flow [8, 24] of features between the *processed* frame for which results are obtained from the server and the *current* frame viewed by the user. To aid this tracking, in addition to returning labels and bounding boxes, the server also returns the feature points for recognized objects in the *processed* frame. Tracking these feature points will allow us to move the bounding box to the correct location in the *current* frame. Unfortunately, this solution only works if the displacement of the object is small between the frames.

Stage	Google Glass Execution Time (ms)	Mobile Client Execution Time (ms)	Server Execution Time (ms)	Model Memory Usage (MB)	Settings
Road Sign Detection [2]	-	2353 ± 242.4	110 ± 32.1	-	Server uses 4 cores.
Road Sign Feature Extraction	-	1327.73 ± 102.4	69 ± 15.2	0.21/object	Using convolutional neural networks with the BVLC GoogleNet model [29, 26, 51]. Server uses a GPU.
Road Sign Recognition	793.3 ± 102	162.1 ± 73.2	11 ± 1.6	0.03/object	Using linear SVM [18] to classify 1K objects with 4K features. Server uses a GPU.
OpenCV Face Detection	3130.18 ± 800.1	2263.71 ± 478.15	197.77 ± 10.56	0.89	Using the frontal face classifier [56, 55]; the minimum size of the detected face is 30×30 pixels.
FaceSDK Face Detection	-	1129 ± 239.5	92.26 ± 21.79	0.12	Mobile client: Nokia Lumia 928.
Hardware-based Face Detection [4]	-	174.6 ± 70.0	-	-	Mobile client: HTC One M8.
Facial Feature Extraction	309.8 ± 101.2	84.55 ± 25.57	19 ± 3.15	35	Extracting 57K features around 27 landmarks [9, 10].
Face Recognition	2912.3 ± 448	537.8 ± 224.1	41.13 ± 3.11	1.25/object	Using linear SVM to classify 224 objects with 57K features.
Tracking	43.22 ± 9.1	37.7 ± 11.5	1.2 ± 0.4	0	Lucas-Kanade tracking with 3 pyramid levels and 30 feature points [35].

Table 2: Performance of object detection, feature extraction, recognition, and tracking. Unless stated otherwise, the Mobile Client is a Samsung Galaxy Nexus Android smartphone and Server is a Intel Core i7 with 3.6GHz, 4-core CPU. The performance is averaged across 1293 frames with resolution 640 × 480.

The performance of tracking degrades as the displacement increases. Figure 5 shows an example. When the user's view is changing, objects move within hundreds of milliseconds. Hence running merely one object tracking from the *processed* frame to the *current* frame is insufficient to achieve good accuracy (we give experimental results in §7).

To accurately find the location of objects in the *current* frame, Glimpse maintains an active cache of intermediate frames; it runs object tracking through a subset of the cache.

This approach works because tracking works when the object displacement is small, which is generally the case in consecutive frames. The active cache stores all the subsequent frames from the frame that gets transmitted to the server for processing. When the mobile device receives the recognition results from the server, it runs object tracking from the processed frame, through the cached frames, to catch up with the current frame.

Unfortunately, it is not practical to run through *all* the cached frames. Object tracking for a frame takes over 30 ms on today's mobile platforms (see Table 2). Tracking through every frame in the cache does not allow us to catch up to the frame being viewed. For example, suppose we send *frame 1* to the server and the end-to-end delay to obtain results from the server is 1 second. At 30 frames per second, we would have cached 33 frames. Because it takes 38 ms to track an object on a mobile device, we need at least 1140 ms to run through all the cached frames. By the time we produce a result for *frame 30*, the user is already viewing *frame 64*, rendering the results stale. Note that we cannot start object tracking on the cached frames before we obtain results from the server because we need the bounding boxes and feature points for tracking.

Therefore, we have to speed up the processing of cached frames to catch up with the current frame as fast as possible. Discarding frames will speed-up the computation, but might sacrifice trackability. Glimpse uses an adaptive subsampling strategy that takes the observed delay, the device's capabilities, and the changes of the scene into consideration. The method subsamples frames without compromising trackability by solving two problems:

1. How many frames to select from the active cache?
2. Which frames to select?

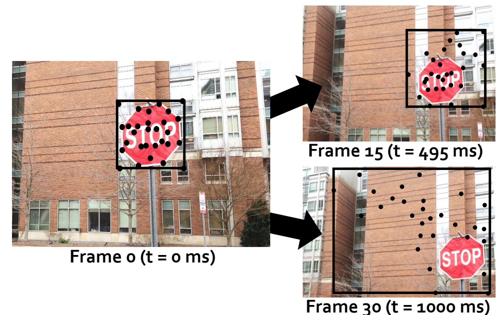


Figure 5: Tracking performance degrades as the displacement of the object increases.

3.1 How many frames to select?

Let f_i be the ith frame captured by the camera. If we send f_i to the server for recognition, we store subsequent frames in the active cache. Assuming the results come back when the client is viewing frame $f_{i+(n-1)}$, we will have cached n frames (f_i to $f_{i+(n-1)}$) in that duration. Our goal is to pick l out of n frames, or $p = l/n$ fraction of frames so that we can catch up without sacrificing tracking performance. A smaller p allows us to swiftly catch up to the current frame, but might degrade performance since we discard many frames; a bigger p ensures reliable tracking, but it takes more time and we would be left with stale results.

The selection of p depends on two factors: (i) the end-to-end processing delay of a frame (i.e., the value of n), and (ii) the execution time e of the tracking algorithm on the client. The lower the mobile's computation capability, the larger the value of e, so p must be small. When the network

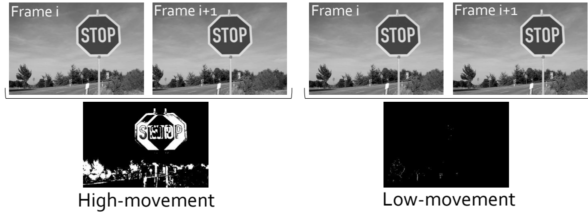


Figure 6: Frame differencing.

delay is low, n is small, and we can select a larger p without degrading trackability.

For any given platform, using a set of videos tagged with ground truth, we simulate different n, e, and p, and pick the p that maximizes trackability. In our experiments, when e is 30 ms or more, $p = 0.1$ works the best for different values of n. But when e is 20 ms or less, the best p depends on n (§7.2). At runtime, we measure e for every tracking call and maintain an average. We directly find n as the number of cached frames. We query a model stored locally with e and n to get p.

3.2 Which frames to select?

Given a sequence of frames $F = \{f_i, ..., f_{i+(n-1)}\}$ stored in the cache and a fraction p, we select $l = p * n$ frames from the cache for tracking. A straightforward approach is to pick l frames at regular intervals, but this method does not give us the best results when objects are moving across frames. For instance, if the object only starts moving after frame $f_{i+(n/2)}$, we can skip frames in the first half.

The selected l frames should capture as much movement as possible, and not have much redundancy between each other. To solve this problem, we first need a metric to characterize movement between two frames. This metric must be easy to compute, lest we lose the benefit of iterating through only a subset of frames.

Glimpse uses a lightweight *frame differencing* function to calculate the "movement" between two frames (Figure 6). First, we convert the frame to grayscale and compute the absolute difference of pixel values $a_{i,j}(x, y)$ for every pixel (x, y) between frame i and frame j, and consider it significant if it exceeds a threshold:

$$a_{i,j}(x, y) = |f_i(x, y) - f_j(x, y)| \tag{1}$$

$$d_{i,j}(x, y) = \begin{cases} 1 & a_{i,j}(x, y) > \phi \\ 0 & otherwise \end{cases} \tag{2}$$

Based on our experiments, we pick $\phi = 35$, a value sensitive enough to capture scene movements and robust to changes caused by noise. We then compute the frame difference (movement metric) $d_{i,j}$ between frame i and frame j as:

$$d_{i,j} = \sum_{x,y} d_{i,j}(x, y), d_{i,j} \geq 0 \tag{3}$$

Computing $d_{i,j}$ is linear in the size of the frame, taking only a few milliseconds on a mobile device; it can be computed when the frame is inserted into the active cache.

Using the frame difference metric, the frame selection problem can now be redefined as follows: given a sequence of frame differences $D = \{d_{i,i+1}, ..., d_{i+(n-2),i+(n-1)}\}$, divide D into $(l + 1)$ partitions, such that the maximum sum over all the partitions is minimized. This is a *linear partition* problem, which can be solved with a dynamic program in $O(ln^2)$ time, by defining $H[n, l]$ as the optimum value of a partition arrangement with n frame differences and l partitions:

$$H[n, l] = \min_{j=i}^{i+n}(\max\{H[j, l-1], \sum_{k=j}^{i+n} d_{k,k+1}\}) \tag{4}$$

Figure 7: Tracked feature points deviate due to changes in the angle of the object.

4. TRIGGER FRAMES

Unlike existing systems [20, 14, 22] that need to send every frame (or as many as the network can handle) to the server, Glimpse reduces bandwidth consumption by sending only certain *trigger frames* to the server to obtain the object recognition hints.

The Glimpse client uses three techniques:

1. It monitors tracking performance and sends frames to the server when the local tracking is about to fail.
2. It sends frames to the server when there is a significant change in the scene.
3. If the scene is constantly changing, it avoids sending too many frames by keeping track of the number of frames in flight.

Detecting tracking failure.

As shown in Figure 7, the tracked feature points would deviate from their correct location and degrade trackability performance as the size, angle, or appearance of the objects changes. When tracking degrades, the client sends a frame to the server to get a new set of feature points to track.

To measure the tracking performance, we make the following observation: since all tracked points are on the same object, when an object moves between frames, the moving distance of those points between frames should be similar. Therefore, we can quantify the tracking performance by measuring these distances between frames. We use the standard deviation of distance of all tracked points between two frames. The larger the standard deviation, the higher the possibility that tracking is failing. We use a threshold of ω_{std} to identify trigger frames, set using experiments to balance between getting new features in a timely way and not sending too many frames.

Detecting scene changes.

In addition to monitoring tracking performance of recognized objects, we should also identify trigger frames when new objects come into the scene. However, it is too expensive to run object detection on the client. To detect if the scene has changed, Glimpse uses the frame differencing technique described in §3.2. We use the formula in Equation 3 to quantify how much the frame has changed from the previously transmitted frame. When the number of "significantly different pixels" between the current frame and the previously transmitted frame is greater than a threshold, ϕ_{motion}, we identify it as a trigger frame.

Limiting the number of frames in flight.

When the scene is constantly changing, the above heuristics could trigger many frames to be sent to the server, causing a network or server bottleneck and also increasing energy consumption. Glimpse limits the number of in-flight frames sent to the server for which results are pending. When there are already i_{max} frames in flight, and we identify a trigger frame, the client waits to receive the results for a frame before sending the next frame[1]. Our experiments show that, having $i_{max} = 1$ frame strikes a good balance between accuracy and resources consumed for various network types (§7).

5. IMPLEMENTATION

We have implemented the Glimpse client for Android smartphones and Google Glass. We also ported the client to Linux for evaluating Glimpse in a controlled setting (§6).

Glimpse Client: The Android implementation of Glimpse uses the OpenCV library (version 2.4.10) [40] written in C/C++ on top of JNI. To speed up the computation, except the dynamic program for frame selection, all the core functions are written in C++, and are invoked using JNI. On Google Glass, running frame differencing on a 640×480 frame using Java takes 220 ms, compared to only 32 ms using C++. We use a separate thread for receiving camera frames and a separate thread for computation (§7.4 describes the detailed energy/overhead measurements). For object tracking, we use the Lucas-Kanade function [35] provided in OpenCV with the default parameters. We declare an object has disappeared if its bounding box is too small (15×15 pixels for road signs [47], and 25×25 pixels for faces), or we do not have enough feature points to draw the bounding box (i.e., the number of feature points is fewer than 4). OpenCV returns frames captured from the camera in both RGB and grayscale format. Because object tracking does not require a color image, we store the frame as a grayscale image in the active cache. We use 640×480 frames and compress each frame into a JPEG image (quality level = 70) before transmission. Each frame is sent using HTTP POST request with multi-part/form-data content type.

Glimpse Server: The server implements the road sign recognition pipeline and the face recognition pipeline (§2.1).

For road signs, we first detect them using boundary, shape, and color [2]. We speed up detection by parallelizing the image processing with four cores. For feature extraction, we train a convolutional neural network model [26, 29] (BVLC GoogleNet Model [51]) using labeled road signs to obtain the computer-crafted features. For classification, we train a SVM classifier with a linear kernel [18] using the features extracted in the previous step. To enable road sign tracking, we use the *good features to track* [49] and return 30 feature points (the top 26 features and the 4 corners of the bounding box) to the client for tracking.

For face detection at the server, we use the Viola-Jones object detector [56, 55] to determine the locations of faces in the frame. In our implementation, we use the Microsoft Face SDK Viola-Jones detector (C#) because it is faster than the OpenCV implementation (Table 2), and generates fewer false positives. For feature extraction, we first locate 27 semantic facial landmarks such as eyes, nose, and mouth in a facial image using a boosted shape regression [9]. We then build the features by extracting multi-scale patches centered at facial landmarks [10] and describe each patch with the local binary pattern (LBP) descriptor. Similar to classifying road signs, we also train a linear SVM model for classifying faces. For face tracking, we return feature points for the 27 landmarks and the four corners of the bounding box computed during the detection and feature extraction stage.

To train the road sign feature extraction and classification models, we use a subset (50%) of the labeled road signs in our dataset (§6). To train the face classification model, we use a training dataset with 224 subjects, with 20 to 40 images for each face. 211 out of the 224 subjects are public figures and the images are collected from the web with URLs from PubFig [30] and CFW [62]. For the subjects featured in our video dataset (§6), we collected images from their Facebook pages. When the server starts, it reads the detection, feature extraction, and classification models from disk into memory for faster processing.

If the client cannot reach the server to send a *trigger frame* because of network failures, we continue to retry periodically. In the mean time, if we are able to track objects locally, we continue tracking them. Currently, Glimpse does not support completely disconnected operation. It informs the user that the service is disconnected and stops annotating objects. To support disconnected operation, we could implement a lightweight, less-accurate object recognition pipeline locally with a small set of features, trained on a few objects. When the client gets disconnected from the server, we could run the local object recognition pipeline. Note that because the local processing will still be computationally expensive with a high response latency, the active cache and trigger frames will remain important.

6. EVALUATION

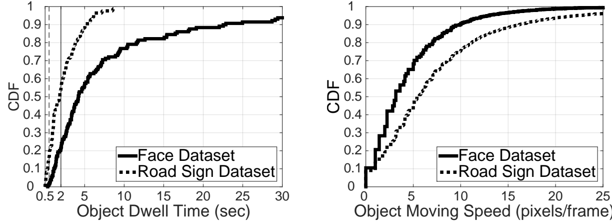
We evaluated Glimpse using real-world data and trace-driven emulation. Emulation allows us to compare Glimpse with other schemes under reproducible conditions. We also ran Glimpse on Android smartphones and Google Glass to measure its resource consumption.

6.1 Data Collection

We collected two datasets to capture different scenarios, and to evaluate Glimpse's ability to recognize and track different objects. For ground truth, we manually inspected every frame, assigned labels, and marked the objects of interest with bounding boxes. Labeling the video frames is a cumbersome manual process; to the best of our knowledge, we are not aware of any publicly available labeled mobile video dataset.

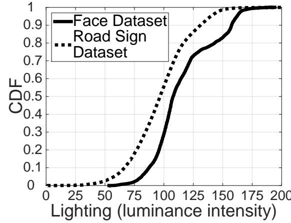
1. **Face Dataset:** We recorded 26 videos with an Android smartphone. The videos are at 30 frames per second at a resolution of 640×480 pixels per frame. The length of each video ranged from 10 seconds to 16 minutes. Scenarios in the video include shopping with friends, chatting in a restaurant, and waiting at a subway station. In all, we have collected and curated 30 minutes of video, with 54,020 frames and 35,824 (non-distinct) faces.
2. **Road Sign Dataset:** We downloaded 4 walking videos recorded with Google Glass on YouTube. Two videos are from the United States, one from England, and

[1]We always send the current frame to the server even if the identified trigger frame was in the past.



(a) The CDF of the dwell time of each object.

(b) The CDF of object moving speed.



(c) The CDF of luminance intensity of frames.

Figure 8: Dataset characteristics.

one from China. In total, we have 35 minutes of video, with 63,003 frames and 5,424 labeled road signs.

Figure 8(a) shows the CDF of dwell time of each object for the two datasets. The dwell time is the total time an object continuously appears in the view of the camera. In the road sign dataset, 50% of the road signs have a dwell time less than 2 seconds, and 90% of the road signs are less than 5 seconds. In the face dataset, 50% of the faces have a dwell time less than 5 seconds, and 90% of the faces are less than 20 seconds. Figure 8(b) shows the CDF of movement of objects between successive frames (in pixels per frame). Our datasets contain objects moving at various speeds. In the road sign dataset, 90% of the signs move less than 15 pixels between two successive frames (the inter-frame time is 33.33 ms). In the face dataset, 90% of the objects move less than 10 pixels.

Figure 8(c) shows the CDF of luminance intensity (0 - 255) of each frame; a smaller value indicates a darker frame. Our dataset contains varied lighting conditions (luminance intensity between 50 to 160). Note that mobile cameras automatically adjust their aperture to avoid excessively bright or dark images.

6.2 Experimental Setup

To conduct comparative and reproducible experiments, we built an emulation framework with a Linux machine running the Glimpse Client, and a Windows machine running the Glimpse Server. The client stores all the 30 recorded videos locally. To simulate a camera feed, each video frame has a timestamp indicating when it should be processed by the client.

We connect the client and server with a crossover Ethernet cable. To emulate different types of networks, we run the Mahimahi tool [37] on the client. Mahimahi takes network measurement traces as input, and delays each packet delivery time based on the per packet delay in the input trace. In our evaluation, we use wireless network traces described in two recent papers [59, 16], which included Wi-Fi, Verizon Wireless's LTE, and AT&T's LTE network. In addition to controlling the network, we also simulate the compute delays for tracking and frame differencing using micro-benchmarks collected from smartphones and Google Glass hardware.

6.3 Evaluation Metrics

To evaluate Glimpse's trackability, we use *intersection over union (IOU)* [14] as the measure. The IOU of object i is defined as

$$IOU_i = \frac{area\,|O_i \cap G_i|}{area\,|O_i \cup G_i|}, \quad (5)$$

where O_i is the bounding box of the detected object, and G_i is the bounding box of object i's ground truth. We consider the object to be *successfully tracked* if $IOU_i > 50\%$ [14] (i.e., accurately located), *and* the label matches the ground truth.

To quantify Glimpse's performance across large number of video frames, we define the following metrics:

- **Precision (%)**, defined as the ratio of the number of objects successfully tracked to the total number of objects detected by the scheme.
- **Recall (%)**, defined as the ratio of the number of objects successfully tracked to the total number of objects in the ground truth.
- **F1 score (%)**, defined as the harmonic mean of precision and recall.
- **Bandwidth Usage (Kbits/s)**, measured as the total number of kilobits transmitted per second between the client and server.

We normalize both precision and recall for each evaluated scheme using the best precision (93%) and the best recall (96%) we can achieve. We obtain the best results when there are no network and server delays and when the client tracks every frame. The trackability in this ideal condition is less than 100% because our ground truth misses a few objects that object tracking is able to correctly track, and object tracking misses a few objects that we are able to correctly tag as ground truth (e.g., objects that are too small). With this normalization, we are able to measure the performance of Glimpse's techniques independent of imperfections in the vision algorithms or labeling.

7. RESULTS

To evaluate Glimpse, we compare it to the following schemes to measure the benefits of the techniques in Glimpse:

- **Server only:** In this scheme, there is no tracking at the client. The client sends frames to the server and outputs the responses as they arrive. We evaluated different server-only schemes, changing the maximum number of frames in flight, and present results for 1 frame and 30 frames in flight (the results for other values are in between these two).
- **No active cache:** Here, the client tracks between the frame sent to the server (*processed* frame) and the *current* frame, without an active cache. It sends a new frame to the server once we receive a response for the previous frame.
- **Active cache only (no trigger frame):** Here, the client maintains an active cache and tracks through frames using our dynamic programming (DP) algo-

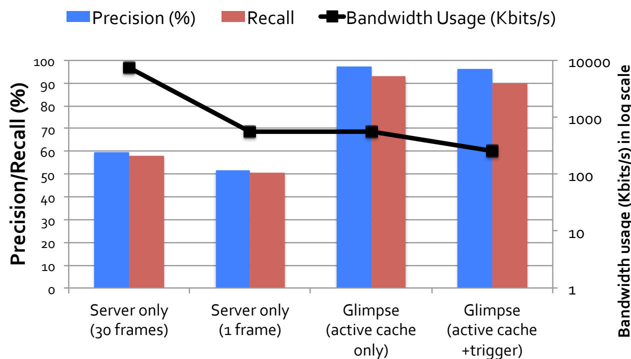


Figure 9: Performance of Glimpse on Wi-Fi for tracking faces. The end-to-end delay (the latency from mobile to server and obtain a response) is 425-455 ms in all schemes, except for 30 frames in flight (542.2 ms).

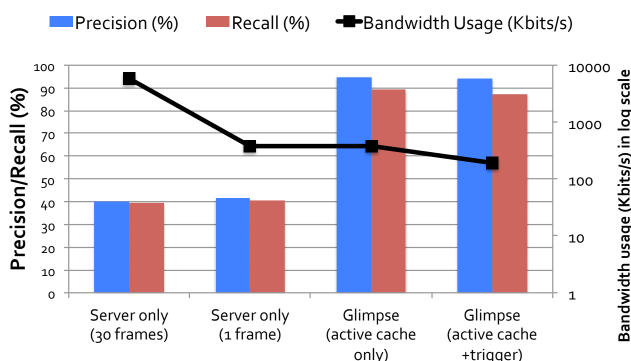


Figure 10: Performance of Glimpse on Verizon's LTE for tracking faces. The end-to-end delay is 656-721 ms in all schemes, except for 30 frames in flight (1102.5 ms).

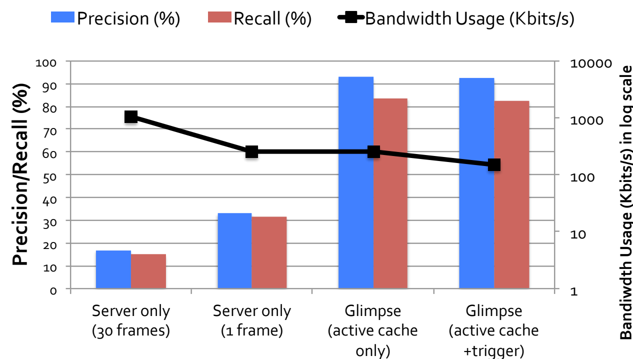


Figure 11: Performance of Glimpse on AT&T's LTE for tracking faces. The end-to-end delay is 927-1041.2 ms in all schemes, except for 30 frames in flight (7391.7 ms).

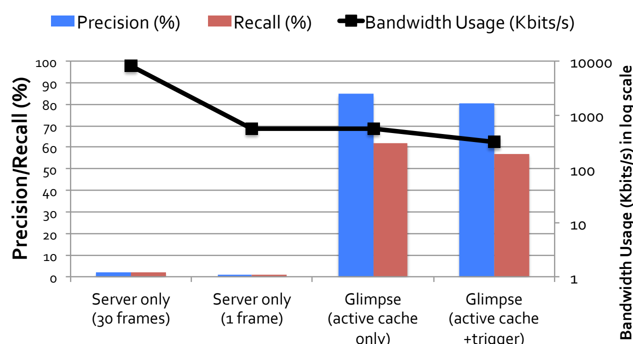


Figure 12: Performance of Glimpse on Wi-Fi for tracking road signs. The end-to-end delay is 510-548 ms in all schemes, except for 30 frames in flight (683.1 ms).

rithm, but there are no trigger frames. The client sends a new frame to the server after receiving response for the previous frame.

In §7.1 we show the end-to-end performance of Glimpse and how it compares to other schemes. In §7.2 and §7.3, we show the benefits of the active cache and trigger frames, respectively. Finally, we evaluate the energy consumption and overhead in §7.4.

7.1 End-to-end performance

We use the network conditions as described in §6.2. Figures 9, 10, and 11, show the end-to-end performance for face recognition on Wi-Fi, Verizon, and AT&T's LTE networks, respectively. Figures 12 and 13 show the end-to-end performance for road sign recognition on Wi-Fi and Verizon's LTE. On the AT&T's LTE network, the network latencies are too high to do real-time road sign recognition. The dwell times of road signs are short (Figure 8), and in most cases, the object moves out of the frame before we get a response from the server.

Significant improvement in precision and recall: Under all network conditions, Glimpse significantly improves the trackability (precision and recall) compared to *Server-only* schemes [22]. The improvement is more prominent as the network delay increases. Even in the fastest network, Wi-Fi, Glimpse improves trackability by 1.8× for face recognition. In the slowest network (AT&T's LTE network), Glimpse achieves a precision of 92.3% (vs. 33%), a recall of 82.6% (vs. 31.8%), an improvement of a factor of 2.8×. For fast moving objects such as road signs (Figures 12 and 13), the improvement is even more significant. Glimpse improves trackability by 71× to 114×; without Glimpse, continuous recognition is non-functional (under 1% precision and recall) for the *Server-only* schemes.

Glimpse's recall decreases as the network delay increases because the dwell time of certain objects become shorter than the end-to-end delay, and hence we miss them. Further, the recall is lower for the road sign recognition compared to face recognition because of the dwell time of road signs are much shorter than that of faces (Figure 8). In Glimpse, when a new object appears in the scene, we identify a trigger frame, but we miss tracking the object till we get a response about that object from the server. Since Glimpse allows only one outstanding frame in flight (§7.3), in the worst case, the discovery overhead could be two round trips worth of frames. When dwell times of objects are longer, we use local tracking to continuously locate the object; but when the dwell time is short (as in the road sign's case), we incur the discovery overhead more often, reducing the recall. If we eliminate the discovery overhead from the recall calculation, the recall increases to 78% on Wi-Fi and 50.2% on Verizon's LTE.

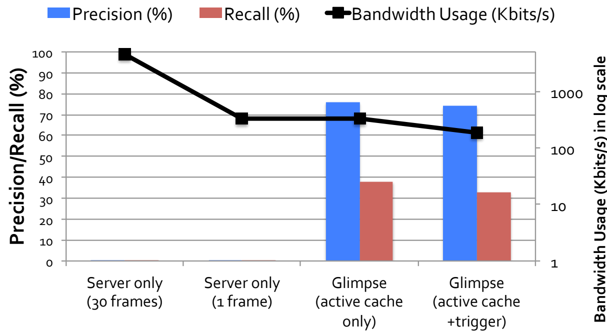


Figure 13: Performance of Glimpse on Verizon's LTE for tracking road signs. The end-to-end delay is 901.1-963.2 ms in all schemes, except for 30 frames in flight (1765.2 ms).

Networks	Schemes	Precision (%)	Recall (%)	Bandwidth Usage (Kbits/s)
Wi-Fi	Server only	56.5	52.2	616.1
	Server only (1 frame in flight)	54.7	52	52.5
	Glimpse	**99.8**	**92.1**	**28.9**
Verizon's LTE	Server only	50.8	47.4	605.2
	Server only (1 frame in flight)	47	44.3	40.2
	Glimpse	**99.4**	**90.7**	**25**
AT&T's LTE	Server only	38.4	31.6	335.8
	Server only (1 frame in flight)	41.3	39	32.1
	Glimpse	**96.4**	**85.5**	**20.7**

Table 3: Glimpse with face detection in hardware.

Significant bandwidth reduction: Figures 9, 10, 11, 12, and 13 also show the bandwidth usage for various schemes. Trigger frames reduce the amount of bandwidth used without compromising precision or recall compared to using only the active cache.

7.1.1 Face detection in hardware

As mentioned in §2.3, some mobile devices [1] are now equipped with face detection hardware. Even with face detection at the client, the recognition operation still needs to be offloaded to the server. If Glimpse's goal is to only detect faces, the face detection hardware can help us reduce the amount of the data transmitted to the server. Instead of sending the entire frame, we can send only the bounding box of faces. Even in this case, Glimpse needs to hide the latency of face detection at the client and the latency introduced by the network and the server.

To show that active cache and trigger frames are beneficial irrespective of where detection happens, we evaluated Glimpse on the face dataset with face detection done on the client. Table 3 shows the results. The baseline scheme sends only the detected faces to the server instead of the entire trigger frame. Glimpse considers the current frame as a trigger frame if (1) tracking fails, or (2) the face detection hardware detects a new face that the system does not recognize.

We measure the hardware face detection latency using the Android FaceDetectionListener [4] API on a HTC One M8 and incorporate it into our emulator. The face detection latency is listed in Table 2, and the number we get is similar to the one reported in [11].

Without Glimpse, even with the face detection hardware, the trackability is poor, and the bandwidth usage remains high since the device does not have the capability to recognize the face, and needs to send all the detected regions to the server for recognition. Glimpse improves the trackability by 1.8× to 2.3×. Since Glimpse tracks the face on the client, face detection hardware helps us detect when new faces appear, reducing the bandwidth usage.

These results demonstrate that regardless of where processing happens, if the processing incurs a latency, Glimpse hides it effectively using its *active cache*. Similarly, *trigger frames* enables the system to reduce the number of frames actually processed thereby saving resources.

Datasets	Networks	Glimpse (active cache only) F1 score (%)	No active cache F1 score (%)
Road Sign	Wi-Fi	71.4	48.5
	Verizon's LTE	50.9	26.1
Face	Wi-Fi	95.1	87.2
	Verizon's LTE	91.9	82.1
	AT&T's LTE	88	78.1

Table 4: The F1 score of Glimpse (active cache only) vs. without maintaining an active cache.

7.2 Benefits of the active cache

Table 4 shows that maintaining an active cache improves the trackability, compared to the *No active cache* scheme, which does tracking without a cache.

The performance of *No active cache* degrades as the network delay increases because it becomes harder to track the object without going through intermediate frames.

For the road sign video, on Wi-Fi, the *No active cache* approach achieves 48.5% F1 score comparing to 71.4% with the active cache, which is an 1.4× gain. For the face video, we also improve the F1 score by up to 12% (AT&T's LTE).

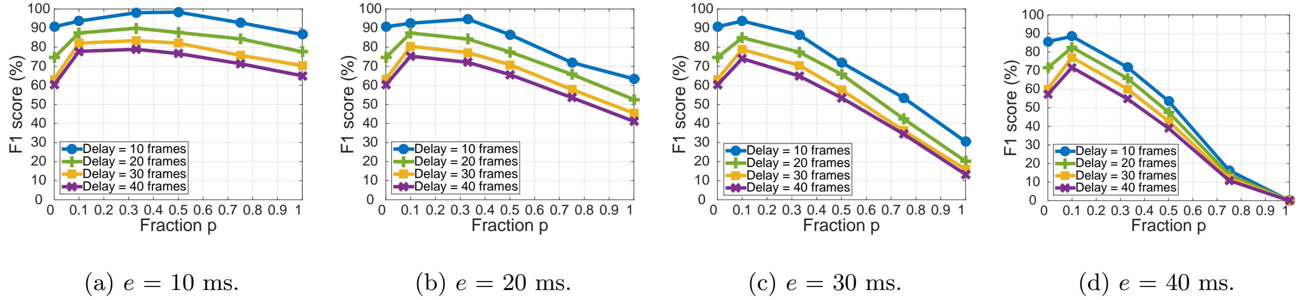
The benefits of the active cache stem from our ability to pick the right frames for tracking. As we show next, simply tracking through all frames in the cache can lead to poor performance.

7.2.1 Picking a subset of frames from the active cache

As discussed in §3.1, object tracking on the client takes tens of milliseconds. By the time the client processes all the frames in the cache, depending on the end-to-end delay, the final tracking result could be stale as hundred of milliseconds would have elapsed.

The fraction of frames that can be processed depends on the execution time of object tracking e, and the number of frames in the cache n. Figure 14 shows the F1 score (y-axis) when we change the fraction of frames picked (x-axis) for different n (lines) and different e (sub-graphs). When e is 30 ms, and n is 30 frames, if we were to pick every frame in the cache for tracking, the F1 score is less than 30%. Similarly, if we do not pick any intermediate frame, the F1 score is less than 65%.

Figure 14 shows that the best fraction depends on e and n. In our experiments, e was between 30 ms and 40 ms; hence, we chose 0.1 (independent of n). As hardware capabilities improve, e will decrease, and the fraction to choose will change with n.



(a) $e = 10$ ms. (b) $e = 20$ ms. (c) $e = 30$ ms. (d) $e = 40$ ms.

Figure 14: Performance of tracking based on the fraction of frames picked from the active cache using both datasets.

Network	DP-based F1 score (%)	Fixed-interval F1 score (%)
Wi-Fi	82.2 ± 7.5	73.3 ± 12.4
Verizon's 3G	77 ± 9.4	59 ± 11.7

Table 5: F1 score for hard videos in the dataset. For this experiment, we selected 50% of all videos that had the worst F1 score. For Wi-Fi, we use both the face and road sign datasets. For Verizon's 3G network, we use only the face dataset.

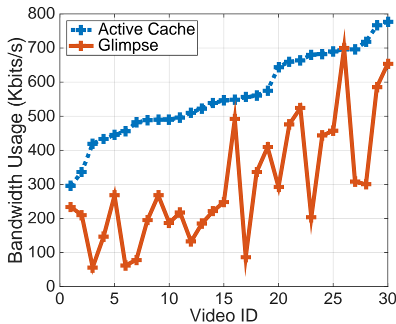


Figure 15: Bandwidth usage of each video by Glimpse and the *Active cache* scheme without trigger frames. The *Active cache* line is sorted by bandwidth usage. The Glimpse line shows the bandwidth usage of the corresponding video.

7.2.2 *Dynamic programming to pick frames*

What is the benefit of the dynamic programming algorithm (§3.2) for frame selection compared to a scheme that simply selects one out of every k frames regularly? Given the fraction of frames to pick from the active cache, we use a dynamic programming approach to capture the object movement instead of picking frames at a regular interval.

We found that for every video in our dataset, the dynamic programming algorithm performs equal to or better than selecting frames at regular intervals. The gain is small for videos with little movement (for them it is immaterial which frames we choose), but high-motion videos are a different story. The aggregate improvement in precision and recall across all videos is only around 4%, but our approach outperforms the fixed-interval scheme on scenarios where there is a lot of movement. Table 5 shows the F1 score for the hard videos in our dataset. For this experiment, we selected 50% of the videos that had the worst F1 score. On these hard cases, the dynamic programming method improves the F1 score by about 12-31% on average depending on the network type.

7.3 Benefits of trigger frames

Under all network conditions (figures in §7.1), *trigger frames* reduce bandwidth by at least 1.5× (2.3× in the worse case) compared to the *Active cache only* scheme, without appreciably reducing precision or recall. We save a factor of 31.3× compared to naively sending every frame to the server.

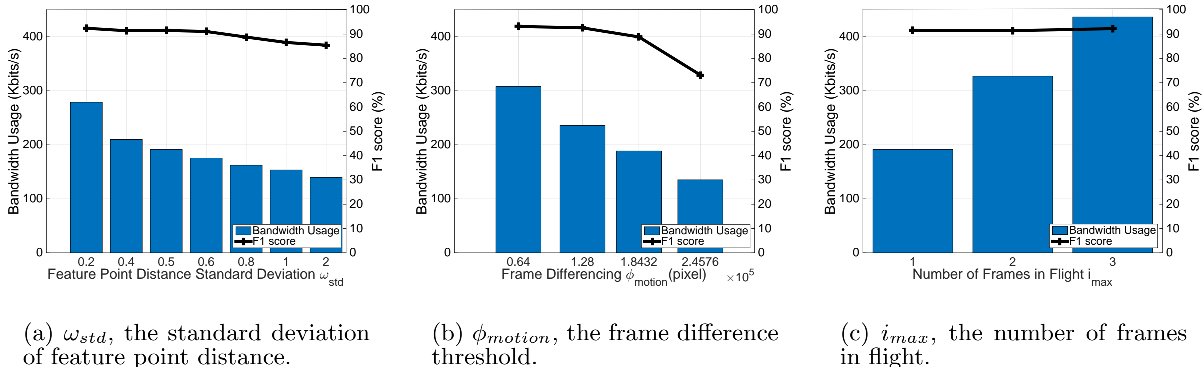
Figure 15 shows the bandwidth usage for each video. It compares the scheme without trigger frames and Glimpse with trigger frames. Glimpse consistently reduces the bandwidth usage. For certain videos, Glimpse provides as much as 6× savings. By transmitting less data to the server, Glimpse also saves energy in the mobile device compared to a scheme without trigger frames. We describe the energy measurements in detail in §7.4.

The bandwidth savings stem from Glimpse's ability to locally track objects without going to the server, and its ability to accurately identify tracking failure and scene changes. We evaluate the importance of both detecting tracking failure and detecting new objects (frame differencing) to identify trigger frames. We compare Glimpse's performance with a scheme that identifies trigger frames only by frame differencing. Though the bandwidth consumed reduces by 53%, the precision and recall also reduce by up to 12%. We get similar performance degradation if we detect only tracking failure without identifying new objects. Hence, detecting both tracking failure and new objects are crucial for identifying *trigger frames*.

We also evaluate the impact of various thresholds used to trigger frames. Figure 16 shows the F1 score and bandwidth usage for varying values of these thresholds. As ω_{std}, the standard deviation of feature point distance, increases, Glimpse becomes more tolerable to deviation of feature points. The number of trigger frames decreases, reducing bandwidth usage, but trackability also decreases because of imminent tracking failures. Similarly, as ϕ_{motion} increases, Glimpse becomes less sensitive to new objects, and we get fewer trigger frames. The bandwidth usage decreases, but the trackability decreases as well because we start missing objects. We pick $\omega_{std} = 0.5$ and $\phi_{motion} = 153600$ (half the number of pixels in a frame) to strike a balance between bandwidth usage and accuracy. Also, allowing more frames in flight has no impact on accuracy, but almost doubles the bandwidth usage. Hence, we use $i_{max} = 1$ in practice.

7.4 Energy consumption

We evaluate the energy consumption of Glimpse by measuring the energy consumption of its components and scaling them in proportion to their usage. We use a Samsung



(a) ω_{std}, the standard deviation of feature point distance.

(b) ϕ_{motion}, the frame difference threshold.

(c) i_{max}, the number of frames in flight.

Figure 16: Bandwidth usage and F1 score for various parameters on identifying trigger frames for the face dataset. We see the same trends for the road sign dataset.

Mode	Power (mW)	Current (mA)	Battery Life (h)
Sleep	56	15.1	122.5
Idle	1009.2	272.9	6.8
Screen on + Frame Capturing	2124.6	574.2	3.2
Verizon's LTE uplink (Active)	2928.2	791.4	2.3
Verizon's LTE downlink (Active)	1737	496.3	3.7
Verizon's LTE (Idle)	1324.6	358	5.2
Wi-Fi uplink	1871.5	506	3.7
Wi-Fi downlink	1289.9	347.8	5.3

Table 6: Energy measurement for various operations on the Samsung Galaxy Nexus with an 1850 mAh battery.

Component	Execution Time (ms)
Frame differencing	7
Active cache (DP frame selection)	1.7
Tracking failure detection	< 1
Tracking	37.7

Table 7: Average execution time of each of Glimpse's components on the Samsung Galaxy Nexus.

Galaxy Nexus smartphone with a 1.2 GHz dual-core ARM processor, 1 GB RAM, and a 1850 mAh battery. The CPU has four operating frequencies: 350, 700, 920, and 1200 MHz.

Table 6 shows the energy consumption for various operations on the smartphone. Without any computation, capturing frames from the camera with the screen turned on consumes 2124.6 mW, and the CPU runs at 350 MHz. When Glimpse is running, it uses two cores, and the CPU raises its frequency to one of the three higher settings, consuming an average of 2947 mW. Transmitting a frame via LTE consumes a total of 2928.2 mW and in Verizon's LTE, the radio tail length [15] is around 10.2 secs. Transmitting one frame on Wi-Fi consumes 1871.5 mW on average. Table 7 shows the average execution time of Glimpse's components. Based on our trace-level emulation, we model the expected energy consumption for running Glimpse.

Table 8 shows the expected battery life when running Glimpse. Compared to a scheme that sends 1 frame every RTT (1 frame in flight), Glimpse improves the battery life

Network	Glimpse battery life time (h)	1 frame/RTT battery life time (h)
Wi-Fi	1.9 ± 0.1	1.4
Verizon's LTE	1.5 ± 0.1	1.2

Table 8: Estimated battery life time.

by 35% on Wi-Fi and 25% on LTE. For Google Glass (Wi-Fi), based on the energy measurement numbers reported in [33], compared to a scheme that sends 1 frame every RTT, Glimpse can improve the battery life by 24%.

8. RELATED WORK

We discuss prior work on network latency hiding, object recognition, and mobile computation offloading.

Network latency hiding: Latency hiding techniques have been widely used as a method for hiding state inconsistencies between distributed nodes. It has been used to mitigate the effects of network latency and provide smooth user experiences in many distributed interactive applications, including remote display [31, 43] and multi-player network games [5, 57]. A common form of latency hiding technique is *dead reckoning*, where clients simulate the states of other hosts using speculation; incorrectly predicted states are rolled back when the actual ones arrive from the hosts.

Glimpse has similar goals, but object movements in video are harder to predict. The results obtained from the server are stale and cannot be directly shown to the user; we need to carefully correct the results to provide a smooth user experience. To the best of our knowledge, Glimpse is the first work that successfully hides network and server latency for object recognition applications.

Object recognition: Object recognition identifies and locates objects in an image. There is a huge body of work on object recognition in the computer vision community, and state-of-the-art methods [52, 10, 21, 19, 32, 3] achieve high precision. Unfortunately, these techniques have significant computation and memory needs, and in practice, realizing them on a mobile platform is difficult. Glimpse can use any of these algorithms, and apply its latency hiding mechanism to hide the processing latency. Besides, much of the advances come from the use of improved object features, which can be easily incorporated into Glimpse for object tracking.

Other work focuses on enabling object recognition on mobile devices with a client-server design. For example, some papers [25, 27] deploy a recognition system that allows clients to upload images to the server for object classification. Other

papers [20, 14] provide continuous object recognition on mobile devices by running the tracker on the mobile client, and the recognition on a server. Another paper [58] proposes a location-aware face recognition framework that uses location information as a hint to reduce the search space of the recognition algorithm. However, these papers consider neither the impact of network and server processing latency on the performance of the system, nor the bandwidth and energy consumed.

Recently, there has been work focusing on optimizing computer vision algorithms to make them usable on mobile devices. For example, Shen et al. [48] optimize the projection matrix for the Sparse Representation Classification (SRC) and implement a fast and robust face recognition system on the mobile device. Glimpse's techniques are orthogonal and can be useful even for these client-only systems to hide the processing delay and reduce the amount of resources consumed.

Object tracking and scene change detection: There are many existing tracking algorithms [60, 17] and scene change detection techniques [39, 63]. Glimpse's active cache can use these tracking methods for catching up with the current frame; similarly, trigger frames can be sent using any of these scene change detection algorithms. Most of these techniques require substantial computational resources, and are not suitable for applications on mobile devices with real-time requirements. We use the Lucas-Kanade tracking [35] and frame differencing for their simplicity, fast execution, and limited resource requirements.

Mobile computation offloading: Resource limitations on mobile devices and the need for responsiveness for compute- and bandwidth-intensive interactive applications have given rise to the idea of delegating work to nearby computers, called "cloudlets" [46], which are one wireless hop away from the mobile device. Gabriel [22, 23] is a system that uses cloudlets for face and object recognition. These cloudlets run the object recognition pipeline and the client ships every frame to the cloudlet. In contrast, Glimpse does not require an extra cloudlet infrastructure, and is readily deployable. Also, the use of cloudlets can still benefit from techniques used in Glimpse to hide the processing latency of compute-intensive recognition tasks (and the network delay in the last hop) and save cloudlet resources by processing only selected frames.

Several systems deal with network variability and device heterogeneity by dynamically determining the most suitable division of work between the cloud and client. Wishbone [38] shows how to take a data-flow graph of operators and partition them across sensor nodes and the cloud using static profiling to determine a good split. MAUI [13] optimizes the energy consumption for interactive applications using an adaptive pipeline partitioning. Its profiler gathers runtime information and finds the optimal partitioning by solving a linear program. Cloud-Vision [50] extends the idea of code partitioning to server farms and aims to minimize the response time given heterogeneous communication latencies and server compute power. Odessa [41] makes offloading and parallelism decisions using runtime profiling with the goal of reducing the total execution delay. Glimpse is complemantary to the above systems. Glimpse's goal is to hide the processing delay irrespective of where the delay occurs and reduce the number of frames processed.

9. DISCUSSION

Generality of Glimpse: Active cache and trigger frames are generic techniques that can be applied to other vision-based applications. The active cache technique can be applied to any vision application that has real-time constraints. For example, a surveillance control system [61] or video analysis application that tracks objects such as cars needs to hide the recognition delay to track the location of objects in real time. The trigger frame technique can be applied to any vision processing system to reduce the resources consumed.

Limitations of Glimpse: Glimpse can have reduced performance when the detected object does not show prominent contrast from the background, and does not have enough salient feature points to enable reliable tracking. Advances in object tracking can be easily incorporated into Glimpse.

Lighting conditions can affect the performance of vision algorithms used in Glimpse. Our dataset covers a reasonably wide range of lighting conditions and we do not notice any performance degradation – for the face dataset, under the Wi-Fi network, the F1 score for the brightest and the darkest videos are 98% and 97%, respectively. However, we believe that performance can degrade in extremely low-light conditions. Advances in cameras and vision algorithms can help improve the performance of Glimpse in these scenarios.

Currently, Glimpse uses a fixed frame differencing threshold (§4) for trigger frames. Our experiments show that a single threshold works well for a wide range of face and road sign recognition scenarios. It is possible that a single threshold might be suboptimal when we include more recognition scenarios, requiring an adaptive threshold [44].

Future work: One direction for future work is to incorporate on-board inertial sensors such as accelerometers and gyroscopes to improve the performance of Glimpse. Inertial sensors can identify and quantify movement, and can help tracking and trigger frame selection. However, inertial sensors only capture the movement of the device and the user, but not the objects that the user is observing. Therefore, a vision-based trigger frame approach would still be required.

10. CONCLUSION

We presented Glimpse, a continuous, real-time object recognition system for mobile devices and wearables. Glimpse captures full-motion video from the camera, recognizes objects, and annotates the images with bounding boxes and labels. Because the vision algorithms for object recognition entail significant computation, Glimpse uses a distributed architecture with the object recognition pipeline running at the server. To hide the processing latencies, Glimpse uses an active cache of video frames on the client and performs object tracking on a subset of frames to correct the stale results obtained from the processing pipeline. To save resources, Glimpse locally tracks objects, and identifies trigger frames by efficiently detecting tracking failure and scene changes. Experiments with Glimpse shows that it achieves high precision and recall, saves bandwidth, and significantly outperforms other schemes.

11. ACKNOWLEDGMENTS

We thank the industrial partners of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support. We also thank the volunteers who helped gather our datasets.

12. REFERENCES

[1] iPhone 6 Face Detection. https://www.apple.com/iphone-6/cameras/.

[2] Road sign detection and shape reconstruction using gielis curves. https://sites.google.com/site/mcvibot2011sep/home.

[3] DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *CVPR*, 2014.

[4] Android camera.facedetectionlistener. http://developer.android.com/reference/android/hardware/Camera.FaceDetectionListener.html.

[5] J. Aronson. Dead reckoning: Latency hiding for networked games. http://www.gamasutra.com/view/feature/131638/dead_reckoning_latency_hiding_for_.php, 1997.

[6] R. Azuma. A survey of augmented reality, 1997.

[7] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). In *CVIU*, 2008.

[8] S. S. Beauchemin and J. L. Barron. The computation of optical flow. In *CSUR*, 1995.

[9] X. Cao, Y. Wei, F. Wen, and J. Sun. Face alignment by explicit shape regression. In *CVPR*, 2012.

[10] D. Chen, X. Cao, F. Wen, and J. Sun. Blessing of dimensionality: High-dimensional feature and its efficient compression for face verification. In *CVPR*, 2013.

[11] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner. Fpga-based face detection system using haar classifiers. In *FPGA*, 2009.

[12] A. Collet Romea, M. Martinez Torres, and S. Srinivasa. The moped framework: Object recognition and pose estimation for manipulation. In *IJRR*, 2011.

[13] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *MobiSys*, 2010.

[14] M. Dantone, L. Bossard, T. Quack, and L. Van Gool. Augmented faces. In *ICCVW*, 2011.

[15] S. Deng and H. Balakrishnan. Traffic-Aware Techniques to Reduce 3G/LTE Wireless Energy Consumption. In *CoNEXT*, 2012.

[16] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *IMC*, 2014.

[17] D. Exner, E. Bruns, D. Kurz, A. Grundhofer, and O. Bimber. Fast and robust camshift tracking. In *CVPRW*, 2010.

[18] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A Library for Large Linear Classification. In *JMLR*, 2008.

[19] V. Ferrari, L. Fevrier, C. Schmid, and F. Jurie. Groups of adjacent contour segments for object detection. In *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2008.

[20] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, and L. Van Gool. Server-side object recognition and client-side object tracking for mobile augmented reality. In *CVPRW*, 2010.

[21] R. B. Girshick, P. F. Felzenszwalb, and D. McAllester. Discriminatively trained deformable part models, release 5. http://people.cs.uchicago.edu/ rbg/latent-release5/.

[22] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *MobiSys*, 2014.

[23] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *MobiSys*, 2013.

[24] B. K. P. Horn and B. G. Schunck. Determining optical flow. In *MIT Technical Report*, 1981.

[25] N. Ismail and M. I. M. Sabri. Mobile to server face recognition: A system overview. *World Academy of Science, Engineering and Technology*, 2010.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[27] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg. Cloud-based robot grasping with the google object recognition engine. In *ICRA*, 2013.

[28] R. Klette, J. Ahn, R. Haeusler, S. Herman, J. Huang, W. Khan, S. Manoharan, S. Morales, J. Morris, R. Nicolescu, F. Ren, K. Schauwecker, and X. Yang. Advance in vision-based driver assistance. In *ICETCE*, 2011.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*. 2012.

[30] N. Kumar, A. Berg, P. Belhumeur, and S. Nayar. Attribute and simile classifiers for face verification. In *ICCV*, 2009.

[31] J. R. Lange, P. A. Dinda, and S. Rossoff. Experiences with client-based speculative remote display. In *ATC*, 2008.

[32] B. Leibe, A. Leonardis, and B. Schiele. Combined object categorization and segmentation with an implicit shape model. In *ECCV*, 2004.

[33] R. LiKamWa, Z. Wang, A. Carroll, F. X. Lin, and L. Zhong. Draining our glass: An energy and heat characterization of google glass. In *CoRR*, 2014.

[34] D. G. Lowe. Distinctive image features from scale-invariant keypoints. In *IJCV*, 2004.

[35] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI*, 1981.

[36] A. Mayberry, P. Hu, B. Marlin, C. Salthouse, and D. Ganesan. iShadow: Design of a Wearable, Real-time Mobile Gaze Tracker. In *MobiSys*, 2014.

[37] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement (demo). In *SIGCOMM*, 2014.

[38] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: ProïňĄle-based Partitioning for Sensornet Applications. In *NSDI*, 2009.

[39] C.-W. Ngo, Y.-F. Ma, and H.-J. Zhang. Video summarization and scene detection by graph modeling. *IEEE Trans. Cir. and Sys. for Video Technol.*, 2005.

[40] Opencv4android. http://opencv.org/platforms/android.html.

[41] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *MobiSys*, 2011.

[42] S. Rallapalli, A. Ganesan, K. Chintalapudi, V. Padmanabhan, and L. Qiu. Enabling physical analytics in retail stores using smart glasses. In *Mobicom*, 2014.

[43] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. In *Internet Computing, IEEE*, 1998.

[44] P. Rosin and T. Ellis. Image difference threshold strategies and shadow detection. In *BMVC*, 1995.

[45] A. Sankaranarayanan, A. Veeraraghavan, and R. Chellappa. Object detection, tracking and recognition for multiple smart cameras. *Proceedings of the IEEE*, 2008.

[46] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE*, 2009.

[47] J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR*, 2012.

[48] Y. Shen, W. Hu, M. Yang, B. Wei, S. Lucey, and C. T. Chou. Face recognition on smartphones via optimised sparse representation classification. In *IPSN*, 2014.

[49] J. Shi and C. Tomasi. Good features to track. In *CVPR*, 1994.

[50] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *ISCC*, 2012.

[51] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CoRR*, 2014.

[52] C. Szegedy, A. Toshev, and D. Erhan. Deep neural networks for object detection. In *NIPS*. 2013.
[53] G. Takacs, Y. Xiong, R. Grzeszczuk, V. Chandrasekhar, W. chao Chen, K. Pulli, N. Gelfand, T. Bismpigiannis, and B. Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *MIR*, 2008.
[54] V. Vaitukaitis and A. Bulling. Eye gesture recognition on portable devices. In *UbiComp*, 2012.
[55] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.
[56] P. Viola and M. J. Jones. Robust real-time face detection. In *IJCV*, 2004.
[57] A. I. Wang, M. Jarrett, and E. Sorteberg. Experiences from implementing a mobile multiplayer real-time game for wireless networks with high latency. *Int. J. Comput. Games Technol.*, 2009.
[58] Z. Wang, J. Yan, C. Pang, D. Chu, and H. Aghajan. Who is here: Location aware face recognition. In *PhoneSense*, 2012.
[59] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.
[60] K. Zhang, L. Zhang, and M.-H. Yang. Real-time compressive tracking. In *ECCV*, 2012.
[61] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. In *MobiCom*, 2015.
[62] X. Zhang, L. Zhang, X.-J. Wang, and H.-Y. Shum. Finding celebrities in billions of web images. *IEEE Transactions on Multimedia*, 2012.
[63] Y. Zhuang, Y. Rui, T. Huang, and S. Mehrotra. Adaptive key frame extraction using unsupervised clustering. In *ICIP*, 1998.