Cachier: Edge-caching for recognition applications

Utsav Drolia[*], Katherine Guo[†], Jiaqi Tan[*], Rajeev Gandhi[*], Priya Narasimhan[*]
[*]Carnegie Mellon University, [†]Nokia Bell Labs
udrolia@andrew.cmu.edu, kguo@bell-labs.com, jiaqit@andrew.cmu.edu, rgandhi@ece.cmu.edu, priya@cs.cmu.edu

***Abstract*—Recognition and perception based mobile applications, such as image recognition, are on the rise. These applications recognize the user's surroundings and augment it with information and/or media. These applications are latency-sensitive. They have a soft-realtime nature - late results are potentially meaningless. On the one hand, given the compute-intensive nature of the tasks performed by such applications, execution is typically offloaded to the cloud. On the other hand, offloading such applications to the cloud incurs network latency, which can increase the user-perceived latency. Consequently, edge computing has been proposed to let devices offload intensive tasks to edge servers instead of the cloud, to reduce latency. In this paper, we propose a different model for using edge servers. We propose to use the edge as a specialized cache for recognition applications and formulate the expected latency for such a cache. We show that using an edge server like a typical web cache, for recognition applications, can lead to higher latencies. We propose Cachier, a system that uses the caching model along with novel optimizations to minimize latency by adaptively balancing load between the edge and the cloud, by leveraging spatiotemporal locality of requests, using offline analysis of applications, and online estimates of network conditions. We evaluate Cachier for image-recognition applications and show that our techniques yield 3x speedup in responsiveness, and perform accurately over a range of operating conditions. To the best of our knowledge, this is the first work that models edge servers as caches for compute-intensive recognition applications, and Cachier is the first system that uses this model to minimize latency for these applications.**

I. INTRODUCTION

Mobile devices have become ubiquitous. Just as their numbers have increased, so have their features - high-definition cameras, microphones and other sensors, multi-core CPUs etc. This has led to people using these devices not only for communication but also to augment their understanding of their surroundings [25]. Recognition and perception algorithms have been a key enabler for such applications [12]. Vision based applications help users augment their understanding of the physical world by querying it through the camera(s) on their mobile devices. Applications such as face recognition, augmented reality, place recognition [11], object recognition, vision based indoor localization, all fall under this category. There are a number of mobile applications available which do one or more of these [4], [2], [8]. Audio based applications such as speech recognition [1], song identification [9], speaker identification [5] are widely utilized by users. In fact, one only has to "look" at things, through wearable devices such as Google's Glass [3] and Microsoft's Hololens [6], to augment their vision of their surroundings.

The ultimate goal of such applications is to be able to point your device towards anything, recognize it, and retrieve information (or media) to augment the user's awareness (or experience), all within a matter of milliseconds. This also implies that these applications have a soft-realtime nature - if the results arrive late, the user might be looking at something else, listening to someone else etc., and the results can be rendered meaningless. Hence, minimizing response time is of primary importance. This is at odds with the fact that these applications are compute-intensive. Moreover, they need access to "big data" to be accurate. If these applications are executed entirely on users' devices, the high computation time would lead to an unusable application. Instead, they rely on offloading intensive tasks to the cloud. Cloud computing has enabled mobile devices to have access to vast amounts of compute resources and data. The devices simply send captured data (audio, images, depth maps, video etc.) to the cloud and receive computed responses. However, the added communication latency still deters seamless interaction, which is vital for such applications.

One suggested approach towards meeting this challenge has been to place more compute resources at the edge of the network, e.g. fog computing [14] and cloudlets [35], both propose placing and utilizing compute resources at the edge, near the user, alongside last-mile network elements. It is proposed that computation can be offloaded from users' devices to these edge resources. This will potentially reduce the expected end-to-end latency for applications.

We propose an alternative to this offloading model. We propose to use an edge server as a "cache with compute resources". Using a caching model gives us an established framework and parameters to reason about the performance of the system. This model makes the edge server transparent to the client, and self-managed - administrators do not need to place content on it manually. An edge server serves a limited physical area, dictated by the network element it is attached to, and thus receives requests from users within that physical area. It can potentially accelerate applications transparently by leveraging this spatiotemporal locality of the requests. It has already been established that document retrieval systems over the Internet, i.e. the World Wide Web (WWW), can reap extensive benefits with caches at the edge [15]. Recognition applications are in fact similar to such retrieval systems. However, unlike requests in the WWW, which use identifiers that deterministically map to specific content being requested, e.g. Uniform Resource Identifier (URI), requests in recognition applications are not deterministic. These applications first recognize the object in the request and use the object's identifier to map to the related content. This is a compute-intensive task.

This property has direct implications on how an edge cache for recognition is designed.

In this paper we first present a model for expected latency in such a cloud-backed, recognition-cache system. We then design a recognition cache, Cachier, that utilizes this model, and show how Cachier efficiently minimizes overall latency for mobile recognition applications. Specifically, we focus on image-recognition applications. Cachier leverages the spatiotemporal locality of requests to store appropriate items locally, that is at the edge, near the user, thus reducing the number of requests that reach the cloud. To the best of our knowledge, this is the first work that models edge servers as caches for compute-intensive recognition applications, and Cachier is the first system that uses this model to minimize latency for these applications. We evaluate Cachier on public datasets under various conditions, and show that it can adapt and lower response time in various operating conditions, without hurting accuracy. We do not propose new computer vision or image-recognition algorithms. Instead we show how we can take existing algorithms and applications, and use edge resources effectively to reduce their user-perceived latency. Also, our approach does not involve the users' mobile devices. Instead we address the interaction between edge resources and backend servers.

In summary, we make the following contributions:

- Model the edge $\leftrightarrow$ cloud interaction as a caching system for recognition applications
- Show how such a recognition cache is different from typical web caches
- Develop Cachier, a cache that minimizes overall latency of image-recognition requests
- Evaluate Cachier on public datasets and a broad range of operating conditions

In the next section we provide background on mobile image-recognition and edge computing. In Section III we model the edge server as an image-recognition cache. We then discuss our approach and optimizations to minimize overall latency in Section IV. Evaluation is presented in Section V, followed by a short discussion in Section VI. Section VII presents related work, and we conclude in Section VIII.

II. BACKGROUND

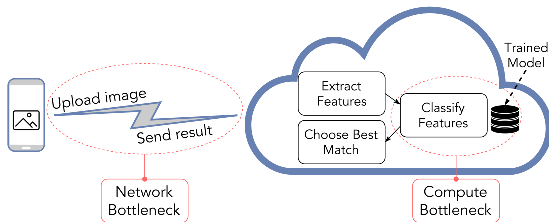


Fig. 1: Typical architecture for mobile image-recognition applications

A. Mobile Image Recognition

Essentially, recognition applications are a kind of information retrieval system, similar to document retrieval systems like the WWW. The system needs to recognize the object in the request for which the user wants information/content, and then use the object's identifier to retrieve the related content.

As shown in Figure 1, within the cloud, the algorithm works in a pipeline manner. A basic overview is as follows.

Extract features. A set of features are extracted from the input image. Features are numerical vectors that describe the image. The class of features that are typically used are called local features - they are extracted from "interesting" points in the image, like corners and edges. They describe the region around these points. A number of such features are extracted from the image to describe it. Collectively, they form a good representation of the image and are effective for recognition purposes. Some notable local features are SIFT [28], SURF [13] and ORB [34].

Classify and match features. Next these extracted features are classified using a trained model. This model is constructed offline, using previously-extracted features from training images of all possible objects that can be queried. The more objects one wants to recognize, the bigger this model will be. Many different types of models have been suggested in image recognition literature, such as brute force nearest-neighbor matching, approximate approaches such as Locality Sensitive Hashing [20] and kd-trees [36], and machine learning approaches such as Support Vector Machines [33].

Choose best match. Once the closest features are found, the object that has the most feature-matches with the request image is chosen as the recognized object. To maintain precision, a threshold of minimum number of matches is set. This can be followed by geometric verification for confirmation that the features in the request image are correctly arranged in space. This also helps in locating the object in the request image. Applications such as augmented reality applications, can use the location information to overlay information or media.

This overall procedure, of extraction, matching with model and verification, is common across vision based recognition algorithms [37].

For mobile image-recognition applications, these stages are typically carried out in the cloud. As shown in Figure 1, the image is captured by the mobile device and uploaded over wireless networks and sent across the Internet to the cloud. The image-recognition procedures are then carried out in the cloud, and the response is returned to the device. These responses are typically in some form of information or content, e.g. annotation strings describing what the user is seeing (e.g. painting recognition), or media to overlay on top of recognized products (e.g. augmented reality).

In such an architecture, overall latency can be broken down into two main factors, (1) Network latency, and (2) Compute time.

1) Network latency: Applications incur this latency since they need to upload significant amounts of data to the cloud, over the Internet, for each recognition request. This latency

can be reduced if the “distance” between the computing entity, currently the cloud, and the mobile device is reduced.

2) Compute time: Image recognition algorithms are compute intensive. The main contributor to latency is the classification of request-image features using the trained model. Moreover, the size of this model, that is the number of trained objects, has a direct impact on the latency. For a large number of objects, the computation time can lead to high latency and hence a poor user experience.

B. Web Caching

Web caching is an established technique of reducing user-perceived latency experienced when retrieving information from across the World Wide Web [38]. Caching information in servers at the edge of the network reduces the time it takes for it to be delivered to users. The same idea is leveraged by content delivery networks [30]. Typically, these caches are not compute intensive - users’ requests contain the specific identifier for the content they want to view and if the cache contains the content tied to the identifier, it is returned to the user, else the request is forwarded to the backend servers. In such systems, the cache has to maximize the amount of relevant content it can serve directly, and minimize the number of requests forwarded to the backend. This is how it can minimize the expected latency for users, reduce network load and backend load.

C. Edge Computing

Edge computing has been recently proposed in multiple incarnations [35], [14]. With the ever-increasing number of Internet-connected devices, it is not feasible for all devices to constantly communicate with a cloud based backend [14]. Edge computing proposes that if more compute resources are placed at the edge of the network, devices can offload compute-intensive tasks to these edge servers and avoid going to the cloud[1]. Instead of just placing content and data near the user, edge computing also provides compute resources at the edge.

In this paper, we show how we can take the idea of caching at the edge, leverage the compute resources now available at the edge, and create an image-recognition (IR) cache.

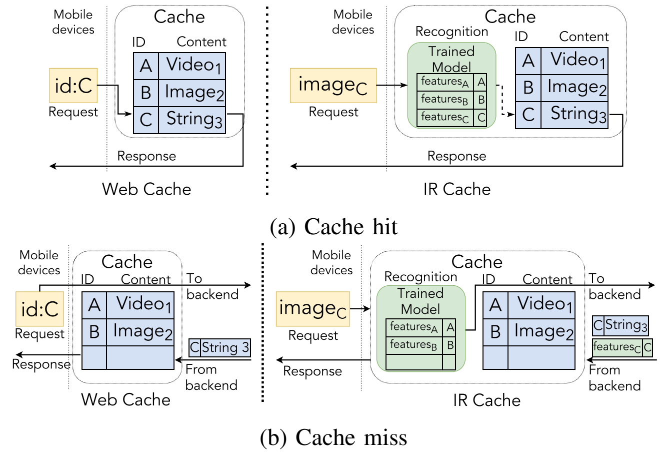
III. Edge-cache for Image Recognition

We propose to use principles from caching systems to model edge resources and their interaction with the cloud, instead of the computation-offloading model proposed in literature. We introduced the idea in [21] and review it here.

A. Why use a Cache?

Nearby users’ recognition requests from applications will likely be for similar objects, that is the requests will exhibit spatiotemporal locality across nearby users. For example, in a museum, multiple nearby visitors will seek information about the same paintings in the area they are in, while shoppers in a grocery store will likely seek nutritional information about

[1]In the rest of the paper we use “backend” and “cloud” inter-changeably.



(a) Cache hit

(b) Cache miss

Fig. 2: Request lookup in a web cache and in an IR cache.

packaged foods. The objects for which the users are seeking information will have little overlap, if at all, across such locations. Hence requests from these areas can be recognized using smaller parts of the trained model that are relevant to the respective locations. We propose to cache these relevant parts of the trained models in edge servers. An edge server serves a small physical region, dictated by the network element it is co-located with. This implies that the requests arriving at the edge server are from users in the same physical region, e.g. at the same museum or the same grocery store. Hence it provides the ideal location to leverage the spatiotemporal locality in requests.

Moreover, a caching model comes with the benefits of a framework for reasoning about and analyzing the system, along with known knobs that can be tuned to optimize for different metrics. It provides an inherent hierarchical structure that is well suited for the edge cache and cloud interaction, and is also extendable - more caches can be added in the hierarchy.

B. Expected Latency

The primary metric for mobile image recognition is the user-perceived latency. This end-to-end latency is the duration between capture of image and the presentation of results to the user. This time comprises of the time taken to upload the image over the wireless network and the wired Internet-backbone, processing in the cloud and the time taken for the response to make its way back to the user. In this paper we are concerned with the latency incurred once a request reaches the edge cache. The latency from the edge cache’s perspective is the duration between the reception of a request at the cache and the transmission of the response from the cache. Hence, it does not comprise of the latency due to the wireless network.

High-level operation. The high-level operation of an IR cache is illustrated in Figure 2, alongside a web cache. In a typical web cache, the incoming request has a specific identifier, e.g. Universal Resource Identifier (URI), to identify the content that is being requested. Typically, a consistent hashing scheme can be used to lookup the content using the identifier. This lookup is computationally trivial. On the other

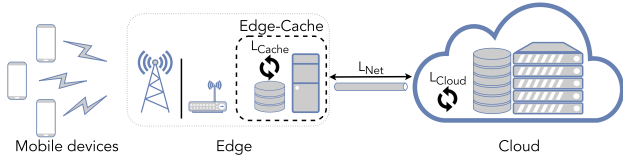


Fig. 3: Location of an edge cache, backed by the cloud.

hand, for an image-recognition application, the request is an image captured by the user. Mapping the image to the content or information being requested is not trivial. This is where the model is used to identify the object within the image, then use the object's identifier to lookup the related content. Hence, the lookup process, which includes object recognition, is computationally intensive. These processes are depicted in Figure 2(a). Given this difference in the internal operation of the two types of caches, the way a miss is handled is also different. On a miss (Figure 2(b)), the web cache forwards the request to the backend, receives the response, stores it locally and sends the response to the user. For the IR cache, along with the response, the cloud also sends the relevant parts of the trained model needed to recognize this request in the future. The cache updates its local model with this. This leads to an increase in the size of the model (number of recognizable objects) in the cache, which is synonymous with the cache size.

To understand how this difference impacts latency, we can model the expected latency for an edge cache (web or recognition), by extending the Average Memory Access Time (AMAT) formula: $H + m * M$, where H = Hit latency, M = Miss latency and m = Miss ratio. For an edge cache, this is given by:

$$E[L] = L_{Cache} + m * (L_{Net} + L_{Cloud}) \quad (1)$$

Where L_{Cache} = cache lookup latency, i.e. time taken by the cache to match a request to the related response, L_{Net} = edge-server-to-cloud network latency, i.e. time taken for a request to reach the cloud after being issued from the edge server, L_{Cloud} = cloud lookup latency, i.e. time taken by the cloud to match a request to the related response, and m = Miss ratio. This is depicted in Figure 3.

Lookup is trivial in web caches, e.g. similar to a hashtable lookup [30]. Lookup latency (L_{Cache}) is small. The size of the cache has negligible impact on the lookup time. This is not the case in an IR cache. Recognizing objects in the request image is compute-intensive, and lookup time is significant. Moreover, when an object is added to the cache, the model size increases, which directly impacts the lookup time [32], [31]. Thus, for an IR cache, $L_{Cache} = f(k)$, $f(k)$ = function of cache size, k. Replacing in Equation (1),

$$E[L] = f(k) + m * (L_{Net} + L_{Cloud}) \quad (2)$$

To investigate this further, let us evaluate an IR edge cache with a Least Frequently Used (LFU) cache-eviction policy, for different cache sizes, deployed as shown in Figure 3[2]. Figure 4 shows the results for that experiment. We see that initially, with a low cache size, the cache reduces overall latency. However, as the cache size (k) increases, even though the hit ratio increases, that is the cache serves more items locally and avoids sending requests to the cloud, the latency increases, and is eventually greater than the latency without a cache.

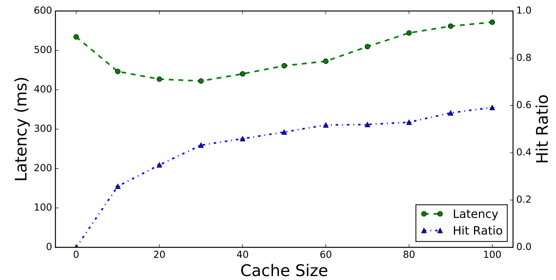


Fig. 4: Overall latency and hit ratio for LFU based IR edge cache, over different cache sizes.

There are two compounding factors that contribute towards this. First, as the size increases, $f(k)$ increases and dominates and increases overall latency. Second, since the computational resources of the edge server are far lesser than the cloud, this rise is stepper still. Note that L_{Cloud} is fixed and low given the resources of the cloud. Unless the network latency is extremely high, for large enough k, $f(k)$ can cause the overall latency to increase. The point at which the inflection occurs is affected by the cache size (k), $f(k)$, cache policy and request distribution. The inflection point is also determined by the network and cloud conditions (L_{Net} and L_{Cloud}). Thus the expected latency formulation needs to incorporate all of these factors.

C. Deconstructing Expected Latency for IR Cache

In the formulation in Equation (2), there are two key parameters, (1) the effect of cache size on latency, $f(k)$, and (2) miss ratio, m. In this section we elaborate on the effects of these parameters for a more explicit formulation of the expected latency.

1) Effect of cache size on lookup latency: As mentioned before, the size of the cache, that is number of objects represented in the trained, local feature set in the edge server, has a direct impact on the lookup latency. This relationship is captured by $f(k)$. This relationship depends on the feature types, the number of features extracted per object image and the recognition/search algorithm utilized to classify the input features. For recognition algorithms, typically, $f(k)$ is a monotonically increasing function.

2) Miss ratio: For a typical web cache the miss ratio, m, depends on the cache size, the cache replacement policy and the underlying request distribution. A miss in such a cache happens only when the item being requested is not in the

[2] More details can be found in [21].

cache. However, there is another kind of miss in an IR cache, a "recognition" miss. Given the nature of the lookup in an IR cache, misses can happen even when the item being requested *is* in the cache. This can happen when the request image contains an object that is not recognized by the cache, even though the model in the cache is trained to recognize that object. The cache cannot distinguish between a "genuine" miss (object's features not present in cached model) and a *recognition* miss. The request needs to be sent to the cloud in both cases. We believe that such cache misses are bound to happen and need to be incorporated in the formulation of the expected latency.

We first simplify m to get a better understanding of the miss ratio. We can write $m = 1 - P(hit)$, where $P(hit) = P(recognized \cap cached)$ is the probability of a cache hit, that is a query being recognized successfully when the corresponding object is in the cache. $P(recognized \cap cached) = P(recognized|cached) * P(cached)$, where $P(cached)$ is the probability that a randomly chosen object is in the cache, and $P(recognized|cached)$ is the probability of correctly recognizing the corresponding object of a request given that the object is in the cache. Let us deal with these two entities separately.

$P(cached)$ is essentially the hit ratio for a typical web cache and as mentioned earlier, depends on the cache size, the cache replacement policy and the underlying request distribution.

$P(recognized|cached)$ is the probability that a randomly selected query will be recognized given that the queried object is in the cache, that is given that the model in the cache is trained to recognize this object. This is also known as *recall*. It is the fraction of correct responses out of all positive queries, that is queries whose corresponding object the model is trained to recognize. This incorporates the accuracy of the matching algorithm into the miss ratio. It depends on the algorithm used and on the size of the model being queried (the cache size in the edge cache's case). As the number of objects in the cache increases, the accuracy tends to drop, that is it is more difficult to precisely recognize the correct match when there are many options to choose from. Given this dependence, we denote this as $recall(k)$.

Putting the formulation of miss ratio back in Equation (2),

$$E[L] = f(k) + (1 - recall(k) * P(cached)) * (L_{Net} + L_{Cloud}) \tag{3}$$

Equation (3) formulation presents a detailed model of the expected latency in a cloud backed IR cache. We started with the high-level AMAT formula and extended it by adding the details for each term in the formula, to represent how they manifest in an edge cache for image recognition. This model presents an alternative perspective to look at edge servers, along with the computation-offloading model.

IV. Cachier's Approach

In the detailed model presented above, we see the cache size (k) as an effective knob that we can tune to minimize the latency of requests in recognition applications. We can now design a system that leverages this formulation to predict expected latency for different k by measuring and estimating the different unknowns in Equation (3). Figure 5 shows the architecture of this system, Cachier. It contains different modules that work in concert to dynamically estimate latencies for different cache sizes and choose the one that minimizes latency. We first explain our intuition about how Cachier will provide a boost to recognition applications and then describe our estimation, profiling and optimization techniques.

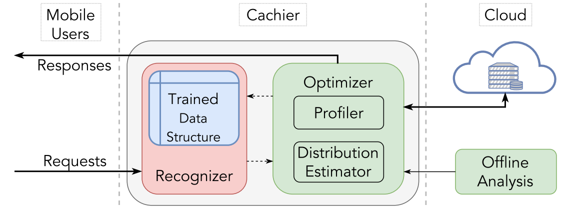


Fig. 5: Internal architecture of Cachier in the edge server. The green modules encapsulate our main contributions.

A. Leveraging Locality

As mentioned earlier in Section III-A, recognition applications will likely exhibit spatiotemporal locality that can be leveraged through caching in edge servers. Instead of a fixed cache size in the edge server, Cachier can dynamically adjust the cache size to store a model for only the most probable requests and reject others even though there is memory/disk space to store a larger model. By keeping the cache size small and relevant, Cachier can cut down on compute time on edge servers. To decide what is the best cache size, Cachier compares the estimated cost of local processing versus the measured cost offloading to the cloud, by using the expected latency formulation, Equation (3).

B. Estimating Effects of Cache Size

As we see in Equation (3), there are three unknown entities that are affected by the cache size, (1) $P(cached)$, the probability that a randomly chosen object is in the cache, (2) $recall(k)$, the probability that a request is correctly recognized given the corresponding object is in the cache, and (3) $f(k)$, the compute time of matching a request to a known cached object. If Cachier changes the cache size to control the latency, it will need to account for changes due to these entities. We look at each in the above order and show how we can estimate these.

C. Estimating $P(cached)$

As we mentioned earlier, $P(cached)$ is essentially the hit ratio for a typical web cache and as mentioned earlier, depends on the cache size, the cache replacement policy and the underlying request distribution. Given the high likelihood of spatiotemporal locality, we start with the Least Frequently Used (LFU) policy and modify it to be dynamically adaptive. LFU eviction policy evicts the least frequently used item in the

cache when the cache is full. Conversely, this policy keeps the most frequently occurring requests in the cache. Now suppose that $k = 1$, then the cache only has the most popular object, and hence the probability of a cache hit is the probability of a request being for the most popular object. Let us now extend this notion. Let j denote the rank of popularity of an object, $j = 1, 2, ..., N$, 1 being the most popular. Let $request_j$ denote a request for an object with popularity rank j. We can then say:

If $k = 1$, then $P(cached) = P(request_1)$
If $k = 2$, then $P(cached) = P(request_1) + P(request_2)$
If $k = n$, then $P(cached) = \sum_{j=1}^{j=n} P(request_j)$

Essentially, under this policy, the probability of a hit in a cache of size k is the probability that the request is for one of the k most popular objects. Thus, to estimate $P(cached)$ the request distribution needs to be dynamically estimated, since that is not available beforehand. The request distribution also captures the spatiotemporal locality of the requests.

Estimating request distribution. The request distribution is modeled as a multinomial probability distribution, $P(request_i) = p_i, i = 1, 2, ...N$, and $\sum p_i = 1$, where N is the total number of objects that can be requested.

Such a distribution can be estimated using MAP (maximum a posteriori) estimation. This requires the system to track the number of times each object i is requested, which is represented by M_i. Using this, the estimate is given by $\widehat{p_i} = \frac{M_i + \alpha_i}{\Sigma M_i + \Sigma \alpha_i}$. α_i is the Dirichlet prior, which is used to provide a prior to avoid overfitting to incoming data, especially when enough data has not been collected. This is generally set to 1, to convey that all objects are uniformly distributed. However, Cachier also uses α_i to insert different priors if it needs to, and we discuss this issue in Section IV-G.

The computation to estimate the request distribution and hence $P(cached)$ is carried out by the Distribution Estimator shown in Figure 5. On receiving a request, the object in the request is first recognized, either by the Recognizer, using the local model, or, on a miss, by the cloud. Once the request is matched to a known object, the Estimator updates the respective object's counter. When a $P(cached)$ estimate is needed for optimization, it calculates the probabilities using the counters and MAP estimation.

D. Offline Estimation of $f(k)$ and $recall(k)$

The effect that changing the cache size would have on the latency and accuracy of the recognition algorithms at runtime needs to be estimated. The compute time and accuracy of the recognition algorithms depend on the choice of the algorithm, the kind of objects they are supposed to identify and the size of the trained model, that is the number of objects the algorithm is trained to recognize. Changing the cache size effectively changes the size of the trained model. Hence, essentially, the $f(k)$ (latency) and $recall(k)$ (accuracy) need to be estimated under different model sizes.

Given a classification algorithm and a dataset, an estimate for $f(k)$ and $recall(k)$ can be found through offline analysis. We present our detailed evaluation of two algorithms, on two public datasets. The estimates generated from the offline analysis can then be used at runtime to predict the recognition algorithm's contribution to overall latency under different cache sizes.

1) Setup: We estimate $f(k)$ and $recall(k)$ for two algorithms, the basic brute force algorithm (BF) and multi-probe Locality Sensitive Hashing (LSH) [29]. The trained models created by these algorithms are quite different.

BF's model. This simply stores a list of extracted features for each training object, and at runtime finds the nearest neighbor of each request image's feature in this list.

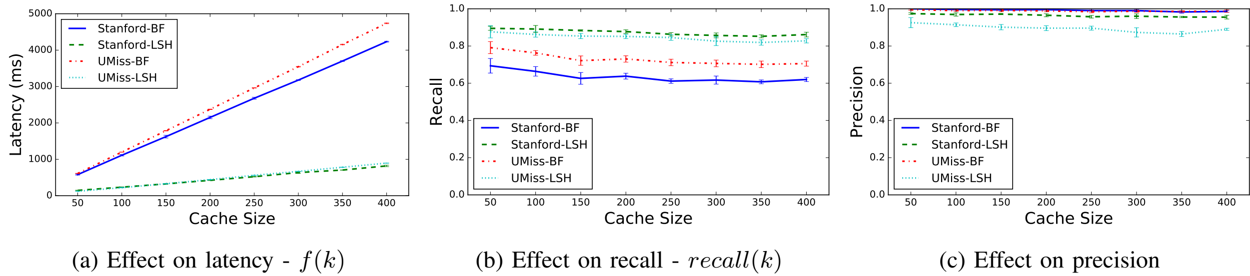
LSH's model. This is an approximate nearest neighbor search algorithm. It creates hash tables using hash functions such that two similar features will most likely be hashed into the same bucket. The training features are hashed and stored in these hash tables, and at runtime the same procedure is applied to request images' features to find which trained features they are closest to.

The binary ORB [34] features are extracted and used for the recognition.

We estimate the $f(k)$ and $recall(k)$ for these algorithms on two datasets, which we call Stanford [16] and UMiss [39], each consisting of 400 objects like paintings, CD covers, book covers, movie posters, magazines, and food packaging. Each dataset has one perfect training image for each object and multiple test images taken from mobile devices. Some examples of training and test images are shown in Figure 6. The models based on features from the training images are the ones that are first stored in the cloud and then in the caches, while the test images are used as the request images sent from the mobile device.

2) Procedure: We use the datasets to profile the two algorithms for increasing number of objects in a pre-populated cache. Along with the perfect image for each object, a request image is added to the training set for better recall. The compute time, recall and precision for looking up request images is measured. The image requests consist of 100 test images of objects that are known to be in the cache (relevant requests) and 10 test images of objects that are known not to be in the cache (noisy requests). The mean and standard deviation of the measurements using two algorithms on two datasets are recorded for each cache size in Figure 7. The estimation of the polynomials presented in Table I and Table II is done through

Fig. 6: **Top:** Training images from the dataset. **Bottom:** Corresponding query images in dataset, taken by mobile devices.



(a) Effect on latency - $f(k)$ (b) Effect on recall - $recall(k)$ (c) Effect on precision

Fig. 7: Offline Analysis of two datasets, using two algorithms. Note that high precision is important to maintain accuracy.

	LSH	BF
Stanford	$0.90 - 0.00014k$	$0.66 - 0.00013k$
UMiss	$0.88 - 0.00013k$	$0.78 - 0.00021k$

TABLE I: Estimated $recall(k)$

	LSH	BF
Stanford	$51.14 + 1.87k$	$60.52 + 10.42k$
UMiss	$6.18 + 2.21k$	$21.88 + 11.77k$

TABLE II: Estimated $f(k)$

regression analysis. The presented estimates are linear because higher order terms are insignificant, which is apparent from the measurements in Figure 7 as well.

3) Recall and precision: Although both high recall and high precision is desirable, high precision is especially important, that is we want all the definite answers from the cache to be correct, since we do not want the presence of a cache to negatively affect the accuracy. An “unknown” answer from the cache is treated as a cache miss and is forwarded to the cloud. We can see in Figure 7(b) that the recall dips slightly with increase in cache size and the trend is similar across the datasets. On the other hand, precision is constantly high (>90%) across cache sizes. The estimated polynomial for $recall(k)$ for the different algorithms and datasets are listed in Table I.

4) Analyzing f(k): We see that LSH is certainly much faster than brute force. The trend is very similar for both datasets (Figure 7(a)), as the cache size (k), that is the number of trained objects in the model, increases, $f(k)$ increases. LSH is the default algorithm used in Cachier. The estimated polynomial for $f(k)$ for the different algorithms and datasets are listed in Table II.

These estimates are fed into the online optimizer module of Cachier, which uses it to predict the effects on overall latency for different cache sizes.

E. Cloud and Network Profiler

The only remaining component in Equation (3) is the latency penalty on a miss, namely $L_{Net} + L_{Cloud}$. The Profiler module in Cachier tracks this online and keeps a moving average filter to even out noise. For every miss in the cache, the Profiler measures the time between issuing the request to the cloud and receiving a response from it. This is useful because if the cloud is overloaded with requests and is taking longer to compute responses then this gets incorporated in the measurements and allows Cachier to adapt dynamically.

F. An Adaptive Cache-inclusion Policy

Now we have a way to estimate the effects of changing the cache size on each of the components in the Equation (3). The Optimizer module brings these techniques together to find a cache size that minimizes the expected latency. It realizes the formulation by using the estimated $f(k)$ and $recall(k)$ polynomials, the estimated request distribution to find $P(cached)$, and the measured miss penalty. This formulation then becomes a function of only k. Given the simplicity of the function, it is not expensive to compute the expected latency for a given k and it takes less than 1ms.

Periodically, Cachier searches for a cache size that minimizes the formulation given in Equation (3) using gradient descent. It essentially is a dynamic, adaptive, cache-inclusion policy - it decides dynamically how many of the most frequently occurring requests should be included in the cache. Once it finds the right cache size k, it places the top k items in the cache, that is the model in the cache can now recognize those top k objects. The Recognizer then uses this updated model to execute future recognition requests.

G. Optimizations

We now discuss some system-level optimizations that we use to design a more efficient system.

1) Learning from previous operations: If Cachier is powered up for the first time, it has no historical data and starts from scratch, with uniform Dirichlet priors for the request distribution. However, once operation starts, Cachier periodically logs the request distribution to disk, and the next time it boots up, it can use this history as the priors for the estimation. The priors are inserted as the α when finding the MAP estimates of the request distribution (Section IV-C). This lets it predict the stable cache size right after startup, without waiting to collect data from requests. This does assume that the request distribution does not change significantly. We discuss this assumption further in Section VI.

2) Lazy feature-fetching: Figure 2(b) depicts that on each cache miss, the request is forwarded to the backend in the cloud and the reply contains both the response for the user and the relevant features for the model in the IR cache. An IR cache miss occurs if the cache doesn’t recognize the object in

System	Features Extracted	Recognition Algorithm	Verification	Cache Parameters
Cachier	ORB	Multi-probe LSH	Geometric	$f(k)$, $recall(k)$ estimated polynomials
LFU Cache	ORB	Multi-probe LSH	Geometric	Cache Size = 400
No Cache (Cloud only)	ORB	Multi-probe LSH	Geometric	N/A

TABLE III: System configurations

the request, but it still might have the relevant features in the set. If features are fetched from the cloud in such cases, communication bandwidth will be used needlessly. To optimize this interaction in Cachier, the backend only returns the object identifier and the response to Cachier. Cachier checks if the identifier is present in its local set. Only if it is not, it requests the features of the object using the identifier returned by the backend. This eliminates the unnecessary traffic between the edge and the backend.

Additionally, Cachier also uses a holdback queue for feature requests sent to the cloud. When a feature request is sent to the cloud for a missing object, the request is also inserted into the holdback queue. Before sending any feature requests, the holdback queue is checked, and if a request for the features of the same object is found, the newer request is dropped, since it is redundant. This is useful when successive request images contain the same object but that object is not in the cache. It may happen that the requests come in quick succession, and the features of the missed object have not yet made it to the cache from the cloud. Then the second request will also miss in the cache, which will generate a request for the same features. However, the holdback queue will prevent the second request from actually being sent to the cloud, thus reducing bandwidth consumption.

V. Evaluation

The primary focus for evaluating Cachier is to measure the reduction in latency achieved under different operating conditions and its impact on accuracy, if any. We evaluate two other systems alongside Cachier, an LFU based edge cache and another system without an edge server at all, so all requests go straight to the cloud. For each of our experiments we show how Cachier compares to them.

The evaluation is based on visual search/augmented reality application scenarios, where users use their devices to seek more information about objects by capturing images. For example, in a museum, visitors may seek information about the paintings, or shoppers in a grocery store may seek nutritional/allergen information about packaged foods. The evaluation setup and procedure are designed to emulate such an application.

Setup. The experimental setup is modeled after the edge server deployment shown in Figure 3. Only the edge server and cloud are part of the experimental setup since no analysis is required for the device-to-edge-server segment for the scope of this paper.

A powerful PC functions as the edge server (4 cores, 8 Hyper-Threads, 8GB RAM), same as the one used in offline estimation in Section IV-D, and is directly connected to a server which serves as the cloud (12 cores, 24 Hyper-Threads, 32GB RAM). `netem` [7] and `tc` [10] are used to control the bandwidth and round-trip times (RTT) between the two machines.

The image recognition and system configuration parameters for each evaluated system are listed in Table III. Note, the cache size for LFU Cache is set to 400, that is the total number of objects it can possibly see. Typically, this is the best case scenario for a web cache.

The UMiss dataset is used for the object dataset and requests. It has a total of 400 objects in it. We use 2 images per object, and 1600 features per image, which leads to a grand total of $\sim$1.2 million features. The training model in the cloud is trained with all of these features.

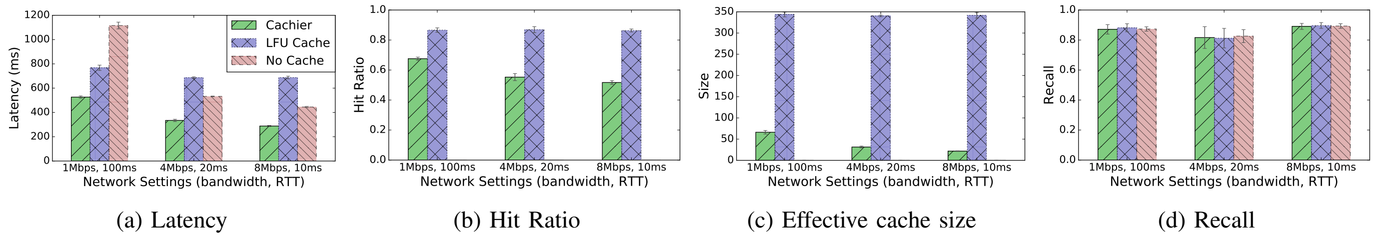
The Zipf distribution is used to model request distribution for such an application. Web caching systems have shown that this distribution accurately models request distribution from users across the Internet [15]. Although this does not necessarily apply for image recognition applications, with the lack of real-world traces of image recognition requests, a Zipf distribution for the requests provides the best approximation.

Procedure. For each experiment, we run 4 iterations. In each iteration, 5000 queries from the dataset are issued to the system. Metrics measured for each request are, (1) the request-response latency, (2) cache hit or miss, (3) correct response or not and (4) the effective cache size, which records the number of items the cache serves by the end of the experiment. In each experiment we report the mean and standard deviation across the iterations.

A. *Effect of Network Bandwidth and Latency*

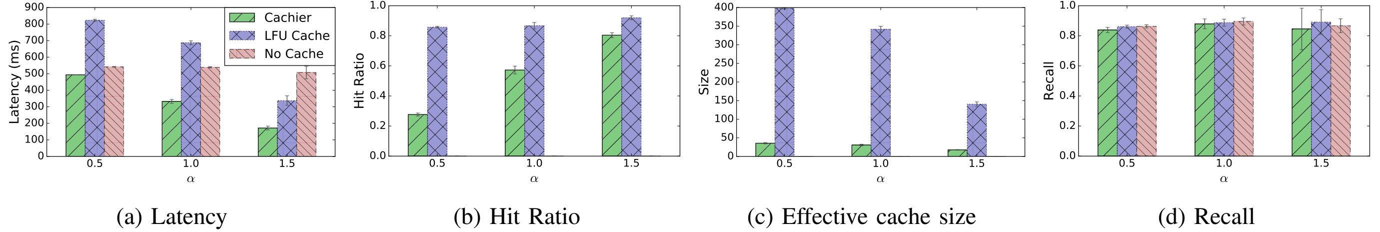
The primary reason of deploying computing at the edge is to minimize end-to-end latency experienced by users. This is directly affected by network conditions between the edge and the cloud. To evaluate its effect we run an experiment with different network settings. Each setting is defined by the network bandwidth and round-trip time. The experimental settings have been setup to mimic measurements presented in [22]. The results are presented in Figure 8.

Reduction in latency. We see in Figure 8(a) that including Cachier in the system reduces latency by almost half when network is slow (1Mbps and 100ms RTT). The LFU cache, also reduces latency in this setting, but not as much as Cachier. The LFU Cache does not benefit from faster network conditions. On the other hand, Cachier adapts to the network conditions and leverages it. It reduces the latency even in fast networks. We can see how it adapts and leverages the fast network by looking at the hit ratio (Figure 8(b)) and effective cache size (Figure 8(c)). We see that as the network performance increases, the hit ratio and effective cache size for Cachier decrease. It reacts to the fast network and decides



(a) Latency (b) Hit Ratio (c) Effective cache size (d) Recall

Fig. 8: Evaluation of Cachier and comparison with a typical LFU Cache and a cloud-only system, for different network conditions. The α for the Zipf distribution governing the requests was fixed to 1.0.



(a) Latency (b) Hit Ratio (c) Effective cache size (d) Recall

Fig. 9: Evaluation of Cachier and comparison with a typical LFU Cache and a cloud-only system, for different request distributions. The network conditions are fixed at 5 Mpbs and 40ms RTT.

that it is better to incur misses and offload to the cloud, than to increase the cache size and increase computation time at the edge server.

No change in accuracy. As we can see in Figure 8(d), the recall of the application is not affected by the inclusion of an edge cache, and tracks the recall of the cloud-only system (labelled as No Cache in Figure 8 and Figure 9) . The precision also remains the same and has not been included in the figures due to space constraints.

Thus we can conclude that Cachier's network awareness helps it to decide when to do more at the edge and when to let the cloud handle the bulk of the computation. This helps it reduce the latency across a variety of network conditions, without sacrificing accuracy.

B. *Effect of Request Distribution*

Request distribution plays a key role in determining the reduction in latency that the cache is able to achieve. To evaluate Cachier's adaptability, we run an experiment with different request distributions using the Zipf distribution, with different α. A low α leads to a more uniform distribution, which represents scenarios where the requests have little locality. A high α leads to a skewed distribution, which represents higher locality in requests - there are some objects which are much more popular than others. The results of the experiments can be seen in Figure 9.

Reduction in latency. Figure 9(a) shows us that LFU Cache requires considerably high locality in requests to outperform the cloud-only system. Cachier adaptively leverages the locality, and this can be understood through Figure 9(b) and Figure 9(c). When the locality is low, cache size is low, hit ratio drops, that is the cloud does most of the processing. When the locality rises, we see that Cachier counter-intuitively decreases cache size. It estimates that it can serve a large number of the requests with a low cache size, and thus keeps the compute time low. This allows it to speed up the response by 3x compared to the cloud-only system.

No change in accuracy. Again, we see, in Figure 9(d), that the recall of the application is not affected by the inclusion of an edge cache, even though they answer a majority of the requests when locality is high.

We see that over multiple different operating conditions, Cachier always outdoes the LFU Cache and the cloud-only system. However, we must note that, as with any caching system, the performance gain can increase manyfold under right conditions, e.g. if there is even higher locality along with poor network conditions, Cachier can take both aspects into account and provide a boost greater than 3x. The gain might also decrease under other conditions, but Cachier's design will always ensure that it is faster than or equal to going straight to the cloud.

VI. Discussion and Future Work

More than image recognition. We believe that the idea of leveraging spatiotemporal context to accelerate recognition of users' common surroundings through efficient caching at the edge can be effective in other forms of sensed data as well. For example, to translate a speech that an audience is listening, to recognize a song playing in a cafe, or for accurate, indoor localization through depth images. In the future, we plan to generalize our approach to include all such applications.

Limitations. One of the current limitations of Cachier is that it performs best for a static underlying request distribution. If the underlying distribution changes rapidly, Cachier might not be able to catch up since it MAP estimates evolve slowly. In the future, we intend to use an exponential moving average to adapt to changes in the underlying request distribution.

VII. Related Work

Edge computing. The issue of cloud computing hurting latency-sensitive perception and recognition applications has been highlighted multiple times in literature [12], [35], [14]. [22] showcases how offloading such tasks to the cloud increases latency. Compared to the cloud, a "cloudlet", a server one wireless-hop away from the mobile user, provides significant speedup. It makes the case for moving compute resources close to the user to avoid network latency. [14] presents a more extensive computing paradigm called *fog computing*. Fog computing proposes to extend the idea of cloud computing all across the network between the cloud and the user, thus bringing computational resources closer to the user and making the network smarter.

We have built on the concept of "more computing at the edge" proposed by these authors, and proposed how to take the next step once there are computational resources available at the edge, for mobile recognition applications.

Computer vision. There have been advances in the field of computer vision and image recognition algorithms to enable efficient processing locally on mobile devices. Feature extractors and descriptors such as SURF [13] and especially ORB [34] have certainly made it possible to do recognition on the devices. However, as reported in [24], [17], doing it locally does not scale beyond tens of images, and we want to achieve recognition of thousands of objects. Deep learning and neural networks have also made it possible to achieve high recognition accuracy [27], and it has been achieved on mobile devices as well [26], but again the device alone cannot achieve the diverse recognition that is needed [23].

Given that mobile devices alone cannot support recognition across more than a few categories, they still need to offload these tasks to a backend server, typically in the cloud. Cachier is a system that accelerates such applications using edge servers as a recognition cache.

Mobile-cloud computing. There are several systems that have been proposed to offload computing from mobile devices onto backend servers or the cloud to enable more efficient applications. These systems divide the application at different granularities to offload parts of it to the cloud, optimizing for different objectives. [19] offloads specific functions from an application to the cloud and optimizes for energy consumption. [18] on the other hand offloads threads of execution to the cloud to minimize latency.

As we can see, the model used to decrease latency is orthogonal to the model Cachier uses. Instead of deciding which part of computing to offload, which requires changing the application itself, Cachier uses edge servers as recognition caches in-between the devices and the cloud. This does not require any changes in the application on the device, and still accelerates the application.

Mobile-cloud image recognition. Techniques have also been proposed, in the mobile-cloud paradigm, to optimize recognition related applications. Glimpse [17] proposes using a video-frame cache locally on the device that hides the latency of the cloud based pipeline and also tracks objects locally on the device to select frames that actually get offloaded. We believe that Cachier can be transparently used in this system and will drive latencies down further. OverLay [24] creates an inter-object distance map and uses it to shrink the search space. This is similar to our work in that it uses the context to reduce the search space. However, it would be difficult for this system to predict correctly in a dynamic environment. Additionally it needs much more sensed data (accelerometer, compass, gyroscope) apart from the camera itself. Our method of leveraging context and estimating popularity does not make these assumptions.

Web caching. Cachier has certainly drawn inspiration from web caching and CDN systems. These systems place content at the edge of the network and serve users from it to avoid high latencies and overloading a centralized server. [15] shows exactly why this works - requests have spatiotemporal locality. People tend to query for similar web pages and this popularity has a Zipf-like distribution. We claim that recognition based applications present even higher spatial and temporal localities, and caching can drastically reduce their latency. However, the approach to minimize latency is different. CDN systems try to optimize for different objectives, such as byte-hit-rate [30], that is they maximize the hit rate and the number of bytes served per hit. By maximizing this, they minimize latency by going to the remote central server less often. Cachier minimizes latency too but it takes into account the added compute time due to the recognition algorithms and modulates cache size accordingly.

VIII. Conclusion

Users are getting used to querying their environment through recognition applications on their mobile devices and expect seamless operation. To live up to these expectations, applications choose to offload intensive tasks to the cloud. However, this adds network latency and reduces overall responsiveness. Edge computing proposes to meet this challenge by placing compute resources at the edge of the network to avoid going to the cloud for all tasks, thus reducing latency. The state of the art model to use edge resources is similar to how applications use the cloud, that is offload intensive tasks to the edge servers. In this paper, we propose an alternative model to use edge resources. We propose to use them as recognition caches, backed by the cloud. A caching model provides a framework to reason about the performance of the system and also makes the edge transparent to the application itself. We develop a model for expected latency in an image-recognition cache and show how to incorporate the effects of compute-intensive recognition algorithms. We design Cachier, an edge cache that uses this formulation to minimize expected latency by dynamically adjusting its cache size. We show that Cachier can increase responsiveness by 3 times or more. We believe that this is the first work that models edge servers as image-recognition caches, provides a formulation for expected latency, and presents a system that uses this model to minimize latency of image-recognition applications.

REFERENCES

[1] Apple Siri. http://www.apple.com/ios/siri/.

[2] Bing Vision. https://en.wikipedia.org/wiki/Bing_Vision.

[3] Google Glass. https://en.wikipedia.org/wiki/Google_Glass.

[4] Google Goggls. https://en.wikipedia.org/wiki/Google_Goggles.

[5] Microsoft Cognitive Services. https://www.microsoft.com/cognitive-services.

[6] Microsoft Hololens. https://www.microsoft.com/microsoft-hololens/en-us.

[7] netem. https://wiki.linuxfoundation.org/networking/netem.

[8] Nokia Point & Find. https://en.wikipedia.org/wiki/Nokia_Point_%26_Find.

[9] Shazam. http://www.shazam.com/.

[10] tc. http://lartc.org/manpages/tc.txt.

[11] Wikitude. http://www.wikitude.com/.

[12] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan. Advancing the state of mobile cloud computing. In *ACM Workshop on Mobile Cloud Computing and Services*, 2012.

[13] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006.

[14] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[16] V. R. Chandrasekhar, D. M. Chen, S. S. Tsai, N.-M. Cheung, H. Chen, G. Takacs, Y. Reznik, R. Vedantham, R. Grzeszczuk, J. Bach, et al. The stanford mobile visual search data set. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 117–122. ACM, 2011.

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

[18] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *EuroSys*, 2011.

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

[20] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[21] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan. Towards edge-caching for image recognition. In *Smart Edge Computing and Networking (SmartEdge), IEEE International Workshop on*. IEEE, 2017.

[22] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 166–176. IEEE, 2013.

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

[24] P. Jain, J. Manweiler, and R. Roy Choudhury. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 331–344, New York, NY, USA, 2015. ACM.

[25] W. Kelly. Computer vision and the future of mobile devices. http://www.techrepublic.com/article/computer-vision-and-the-future-of-mobile-devices/, August 2014.

[26] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.

[27] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[28] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[29] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[30] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[31] M. Martinez, A. Collet, and S. S. Srinivasa. Moped: A scalable and low latency object recognition and pose estimation system. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2043–2049. IEEE, 2010.

[32] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2161–2168. Ieee, 2006.

[33] E. Osuna, R. Freund, and F. Girosit. Training support vector machines: an application to face detection. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 130–136, Jun 1997.

[34] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. IEEE, 2011.

[35] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[36] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[37] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science &; Business Media, 2010.

[38] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.

[39] X. Wang, M. Yang, T. Cour, S. Zhu, K. Yu, and T. X. Han. Contextual weighting for vocabulary tree based image retrieval. In *2011 International Conference on Computer Vision*, pages 209–216. IEEE, 2011.