FoggyCache: Cross-Device Approximate Computation Reuse

Peizhen Guo
peizhen.guo@yale.edu
Yale University

Bo Hu
b.hu@yale.edu
Yale University

Rui Li
rui.li@yale.edu
Yale University

Wenjun Hu
wenjun.hu@yale.edu
Yale University

Abstract

Mobile and IoT scenarios increasingly involve interactive and computation intensive contextual recognition. Existing optimizations typically resort to computation offloading or simplified on-device processing.

Instead, we observe that the same application is often invoked on multiple devices in close proximity. Moreover, the application instances often process *similar* contextual data that map to the *same* outcome.

In this paper, we propose *cross-device approximate computation reuse*, which minimizes redundant computation by harnessing the "equivalence" between different input values and reusing previously computed outputs with high confidence. We devise adaptive locality sensitive hashing (A-LSH) and homogenized k nearest neighbors (H-kNN). The former achieves scalable and constant lookup, while the latter provides high-quality reuse and tunable accuracy guarantee.

We further incorporate approximate reuse as a service, called FoggyCache, in the computation offloading runtime. Extensive evaluation shows that, when given 95% accuracy target, FoggyCache consistently harnesses over 90% of reuse opportunities, which translates to reduced computation latency and energy consumption by a factor of 3 to 10.

ACM Reference Format:
Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. 2018. FoggyCache: Cross-Device Approximate Computation Reuse. In *The 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18), October 29-November 2, 2018, New Delhi, India.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3241539.3241557

MobiCom '18, October 29-November 2, 2018, New Delhi, India

ACM ISBN 978-1-4503-5903-0/18/10…$15.00
https://doi.org/10.1145/3241539.3241557

1 Introduction

The vision of the Internet of Things has gradually transitioned into reality, with solutions for *smart* scenarios and new IoT devices entering the mainstream consumer market. Many of these upcoming applications revolve around interacting with the environment [5], providing personal assistance to the device user [35], and/or automating what used to be labor-intensive tasks [79].

For example, speech recognition based assistance is now common on smartphones (e.g., Siri, Cortana) and in homes (e.g., Alexa and Google Assistant). These applications sense the environment, process the data on the local device or a backend server, and then act on the result. Since contextual sensing and speech/image recognition are often the key steps for these applications, on-device processing tend to be computationally expensive and energy-hungry in order to achieve sufficient accuracy. While offloading to a server [20] addresses these two issues, latency becomes a concern given the real-time nature of the applications. Much prior efforts have investigated mechanisms to improve specific processing logic of either local computation [49] or offloading [19] of individual applications.

Instead, we pursue an orthogonal avenue. A closer look at these applications suggests there is redundancy in such computation across devices. The same applications are used by multiple devices over time, often in a similar context (e.g., common locations). *Redundancy elimination across devices* can then simultaneously achieve low latency and accurate results. This is a promising optimization technique independent of the specific processing logic. Note that this does not replace or favor either local computation or offloading. The goal is to avoid unnecessary computation, either locally or remotely, as opportunities arise.

However, there is a defining difference between our cases and traditional redundancy elimination. We do not have *exact* matches between the input-output relations. Instead, the most common input types, i.e., images, speech, and sensor reading, come from analog sources. The input values are

rarely exactly identical, but *correlated* temporally, spatially, or semantically, and mapped to the same output.

In other words, we need to gauge the *reusability of the previous output* based on the *similarity of the input*. We refer to this paradigm as *fuzzy redundancy* and highlight the notion of *approximate computation reuse*. Section 2 discusses such redundancy in detail in the context of several motivating scenarios. While traditional, precise redundancy elimination is well studied and widely employed in storage systems [24], networking [70], and data analytic systems [33, 65], existing techniques are ill-suited to the IoT scenarios at hand due to the computation and the approximate matching involved.

Approximate computation reuse involves several steps and challenges: capturing and quantifying the *input* similarity in a metric space, fast search for the most similar records, and reasoning about the quality of *previous output* for reuse.

Step one is straightforward. Existing, domain-specific techniques can already turn these raw input values into feature vectors, and we can then define a metric to compute the distance between them, for example, the Euclidean distance. There are two implications, however. First, leveraging these feature extraction techniques decouples the application specific processing from generic system-wide procedures applicable to any such applications. Second, the app developer can use well-established techniques and libraries, and there is no need to manually annotate or manage the input features.

The other two challenges arise from two fundamental constraints regardless of the underlying scenario: (i) The input data distributions are dynamic and not known in advance, and (ii) similarity in the input does not directly guarantee the reusability among the output.

To address (i), we propose a variant of locality sensitive hashing (LSH), which is commonly used for indexing high-dimensional data. The standard LSH is agnostic to the data distribution and does not perform well for skewed or changing distributions. Therefore, our *adaptive locality sensitive hashing (A-LSH)* dynamically tunes the indexing structure as the data distribution varies, and achieves both very fast and scalable lookup speed and constant lookup quality regardless of the exact data distribution.

For (ii), we propose a variant of the well-known k nearest neighbor (kNN) algorithm. kNN is a suitable baseline since it makes no assumptions about the input data distribution and works for almost all cases. However, kNN performs poorly in a high-dimensional space due to the *curse of dimensionality*, insufficient amounts of data and skewed distribution in the data [10]. Our *homogenized kNN (H-kNN)* overcomes these hurdles to guarantee highly accurate reuse and provides control of the tradeoff between the reuse quality and aggressiveness.

We then incorporate approximate computation reuse as a service, called FoggyCache, and extend the current computation offloading runtime. FoggyCache employs a two-level cache structure that spans the local device and the nearby server. To maximize reuse opportunities, we further optimize the client-server cache synchronization with stratified cache warm-up on the client and speculative cache entry generation on the server.

FoggyCache is implemented on the Akka cluster framework [2], running on Ubuntu Linux servers and Android devices respectively. Using ImageNet [67], we show that A-LSH achieves over 98% lookup accuracy while maintaining constant time lookup performance. H-kNN achieves the pre-configured accuracy target (over 95% reuse accuracy) and provides tunable performance. We further evaluate the end-to-end performance with three benchmarks, simplified versions of real applications corresponding to the motivating scenarios. Given a combination of standard image datasets, speech segments, and real video feeds, and an accuracy target of 95%, FoggyCache consistently harnesses over 90% of all reuse opportunities, reducing computation latency and energy consumption by a factor of 3 to 10.

In summary, the paper makes the following contributions:

First, we observe cross-device *fuzzy redundancy* in upcoming mobile and IoT scenarios, and highlight eliminating such redundancy as a promising optimization opportunity.

Second, we propose *A-LSH* and *H-kNN* to quantify and leverage the fuzzy redundancy for *approximate computation reuse*, independent of the application scenarios.

Third, we design and implement FoggyCache that provides approximate computation reuse as a service, which achieves a factor of 3 to 10 reduced computation latency and energy consumption with little accuracy degradation.

2 Motivation

2.1 Example scenarios

Smart home. Many IoT devices connected to a smart home service platform [22] run virtual assistance software that takes audio commands to control home appliances. The intelligence of such software is supported by inference functions, such as speech recognition, stress detection, and speaker identification [1]. Statistics [8] show that a small set of popular audio commands, e.g., *"turn on the light"*, are often repeatedly invoked. Say two household members issue this command to their respective device in different rooms. Currently, each command triggers the entire processing chain. However, processing both is unnecessary, as the two commands are semantically the same. It would be more efficient if one could reuse the processing output from the other.

Cognitive assistance apps. Google Lens [5] has become very popular, which enables visual search by recognizing objects in the camera view and rendering related information. Key to the app is the image recognition function. Consider a scenario where the tourists near a famous landmark search for its history using the app. Clearly, it is redundant to run the same recognition function repeatedly on different devices for

the same landmark. Although the devices capture different raw images, semantically the images are about the same landmark. If the recognition results can be shared among nearby devices, e.g., by a base station, we can avoid the redundant processing on individual devices.

Intelligent agriculture. Robotic assistance has been deployed to automate agricultural tasks. As an example [79], a fleet of autonomous vehicles move along pre-defined routes to measure and maintain the health of the crops, e.g., watering the crops if they appear dehydrated. Each vehicle captures images and other sensor data (for ambient light intensity, humidity, and temperature) every few meters, recognizes the crop status, and then acts accordingly in real time. The vehicles on adjacent paths record significantly correlated data, and running the same processing function on these correlated sensor data will largely produce the same results. Such repeated processing is unnecessary if the processing outputs can be shared among the vehicles, e.g., through the command center of the robots.

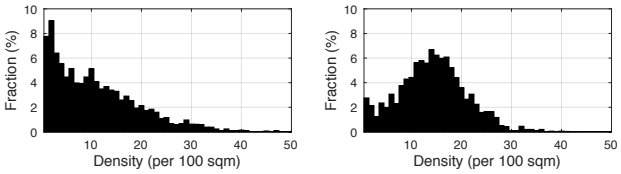
2.2 Fuzzy redundancy

Common to all three scenarios above, the application logic revolves around recognition and inference. There is redundancy in the processing of each application, even when presented with *non-identical* input data. We refer to this as *fuzzy redundancy*. This is due to the *similarity* in the input data, the *error tolerance* of the processing logic, and the *repeated invocations* of the same functions.

Input similarity stems from the same contextual information being captured, such as major landmarks, ambient noise, and road signs. For such information, there is (i) temporal correlation between successive samples (e.g., frames in a video feed), (ii) spatial correlation between what nearby devices observe, and (iii) common objects involved (e.g., traffic lights at each intersection).

Error tolerance arises from a common input-output mapping process, i.e., the input values that are "close" enough are often mapped to the same output. A large number of workloads exhibit this property, where high-dimensional input values (e.g., images of handwritten digits) are mapped to lower-dimensional output values (e.g., "0", "1", ..., "9"), and therefore the possible output values are constrained to a finite set. Learning-based workloads (e.g., recognition, classification, and AI agent) and graphics rendering [55] both exhibit such resilience, and increasingly they have been run in mobile scenarios.

Repeated invocations are manifested in three ways. (i) Given the popularity of some mobile apps, the same app (e.g., Google lens and PokeMon) can be launched by the same device or across devices repeatedly. (ii) Significant correlation exists in spatio-temporal contexts and smartphone usage [78]. For instance, IKEA Place [6], an augmented reality furnishing app, is mostly run by shoppers in IKEA stores.



(a) Outdoors (b) Indoors

Figure 1. Device density distribution from trace [13].

Table 1. Proportion of redundant scenes (%).

Setting	Device density (# devices / 100 m^2)				
	0-10	*10-20*	*20-30*	> 30	*Average*
Indoors	49.63	74.43	99.57	100	64.14
Outdoors	61.01	85.76	99.51	100	82.67

(iii) Mobile applications often rely on standard libraries. This is very common for computer vision (OpenCV [17]), graphics (OpenGL [81]), and deep learning (Model Zoo [3]), which means even different applications can invoke the same library functions.

2.3 Quantitative evidence

To gauge the extent of *fuzzy redundancy*, we estimate the amount of correlated processing for landmark recognition in the aforementioned cognitive assistance scenario. This is measured with the proportion of input images showing semantically equivalent scenes across mobile devices.

First, we estimate the mobile device density distribution by leveraging a WiFi trace from CRAWDAD [13]. Mobile devices frequently scan WiFi access points (APs) to switch or renew their association, by broadcasting a probe request every few seconds. The trace contains probe requests from clients within range of different APs over three months. The AP locations include auditoria, office buildings, malls, and scenic spots. We select the traces at two types of locations, scenic spots and office buildings, for an outdoor and an indoor scenario respectively. The device density (i.e., number of devices per $100m^2$) is calculated by counting the number of distinct devices sending probe requests within a 30-second window. Figure 1 shows the device density distribution at these two locations.

Next, we use Google Streetview API [4] to download streetviews and create an "outdoor input image set" as perceived by a phone camera. The number of images selected is proportional to the corresponding device density distribution measured above. For the sampled images, we count the number of images capturing the same scenes (i.e., buildings, landmarks, and traffic signals) and convert that to a percentage of all images to quantify *the amount of fuzzy redundancy*. Similarly, we use the NAVVIS indoor view dataset [43] for indoors and repeat the above procedures to estimate the portion of redundant scenes.

Table 1 shows the proportion of redundant scenes given different device density. On average, around 64% and 83% of the images exhibit *fuzzy redundancy* for indoor and outdoor

scenarios respectively. The amount of redundancy increases significantly with the device density. This highlights substantial redundancy elimination opportunities to optimize the performance of these contextual recognition based applications.

3 Approximate Computation Reuse

To eliminate *fuzzy redundancy*, we follow the philosophy for conventional computation reuse, i.e., caching previous outputs and later retrieving them instead of computing from scratch every time. However, existing *precise* reuse techniques cannot handle the *approximation* we need.

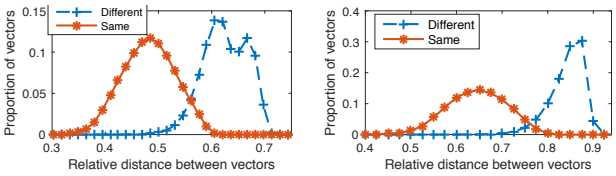
Problems with *precise computation reuse.* Conventional reuse [44, 77] determines the reusability of a previous computation output on the basis of hash-based *exact input matching*. Unfortunately, this is too restrictive for *fuzzy redundancy*, where the input values are correlated but rarely identical. We need to relax the criterion such that computation records are reusable if the input values are *sufficiently similar*.

Challenges for *approximate computation reuse.* Extending exact reuse is non-trivial and requires solving several problems: (i) embedding application-specific raw input data into a generic metric space, (ii) fast and accurate search for the nearest-match records in a high-dimensional metric space, and (iii) reasoning about the *quality of reuse* among the potential search outputs. Challenge (i) can be addressed with well-established domain-specific feature extraction approaches (Section 3.1). To address (ii) and (iii), we propose adaptive locality sensitive hashing (A-LSH, Section 3.2) and homogenized k nearest neighbors (H-kNN, Section 3.3).

Reuse process. Armed with these techniques, *approximate computation reuse* proceeds on a per-function basis. We always turn function input into feature vectors to serve as cache and query keys. Once an inference function is actually executed, a key-value pair is added to the A-LSH data structure. The *value* is simply the function output. When an application invokes a particular function, this triggers a reuse query for that function. We retrieve several key-value pairs from the A-LSH whose *keys* are the nearest-match to the query key (i.e., a feature vector from the *new* function input). Among the *values* of these key-value pairs, we then select thefi nal query result (i.e., the *new* function output) with H-kNN. Section 3.4 discusses the generality of this process.

Crucially, while the input matching is approximate, the ideal output identified for reuse is *precise*, the same as the result from the full-fledged computation, due to the error tolerance discussed earlier (Section 2.2).

Terminology. Thoughout the paper, "input" refers to the raw input data to the inference function or the corresponding feature vectors serving as the cache or query key, while "output" refers to the previously computed results, the cached *value* matching a cache *key* or the reuse query result.



(a) Image data (b) Audio data

Figure 2. Distance distribution between feature vectors of the same and different semantics.

3.1 Application-specific feature extraction

Different contextual recognition applications vary by their input data type (such as images, audio, and text) and inference logic. Therefore, thefi rst step is to embed heterogeneous raw input data into a generic representation while preserving the notion of *similarity*.

There are two implications from this step. First, it decouples application-specific processing from general reuse procedures. Second, it obviates the need for app developers to manually annotate data features.

Assessing similarity is far more challenging than checking for *equality*. Fortunately there are well established techniques to map raw data to multi-dimensional vectors in a metric space. We can then compute the Euclidean distance between vectors to gauge their similarity.

Domain-specific approaches. For *images and videos*, their local and global characteristics can be captured in feature vectors such as SIFT [54] and GIST [61], which have been shown [40] to effectively measure image similarity. For *audio*, MFCC [53] and PLP [38] are widely used to capture acoustic features in compact vectors for speech applications [47].

Autoencoding. More generally, recent Autoencoder techniques [39, 83] use deep neural networks to *automatically* learn state-of-the-art feature extraction schemes for various data sources, including text, images, and speech.

Examples. Figure 2 shows that we can indeed quantifying data similarity with the distance between feature vectors mapped from the raw images and audio samples. The data are randomly selected from three arbitrary classes from ImageNet [67] and the TIMIT acoustic dataset [27]. We use SIFT to turn 256×256 images into 1000-dimension vectors and MFCC to convert 30-ms speech segments (sampled at 16 kHz, 16-bit quantization) to 39-dimension vectors. Figures 2(a) and 2(b) plot the distribution of the distance between pairs of feature vectors, for the image and audio data respectively. The distances are normalized in their respective scale space. We can see that the feature vectors for the same scene (or utterance) are geometrically "closer".

3.2 Adaptive Locality Sensitive Hashing

After turning the raw input data into high-dimensional feature vectors, we need a mechanism to index them for fast and accurate lookup.

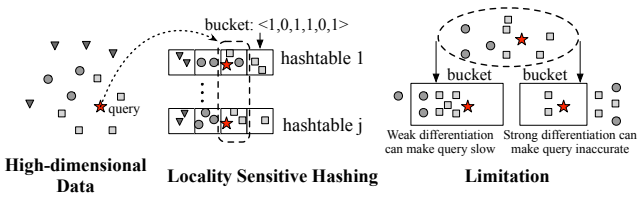


Figure 3. Locality sensitive hashing.

Table 2. Lookup speed comparison (10,000 entries)

Dimension	R-Tree (ms)	LSH (ms)
4	0.018	0.002
64	2.279	0.009
128	6.342	0.011
1024	87.504	0.010

Locality sensitive hashing (LSH) as a strawman. LSH [12] is widely used to search for the nearest matches in a high-dimensional space [28]. The data structure consists of multiple hashtables, each of which employs carefully selected, distinct hash functions and a set of buckets. The hash functions will map similar data to the same bucket in their corresponding hashtables with high probability. The buckets convey a sense of "locality".

Figure 3 shows LSH operations for three clusters of data, represented by three shapes. Ideally, each cluster should be mapped to a distinct, corresponding bucket across hashtables. When searching for the nearest matches, LSHfi rst locates the bucket corresponding to the query input within each hashtable. The entries in all these buckets form a candidate set, from which thefi nal output is selected based on its distance to the query input.

The time complexity for retrieving the nearest neighbors using LSH is $O(n^{\rho}\log n)$, where n is the number of data records indexed and ρ is a variable far smaller than one. In comparison, other spatial indexing data structures such as R-Tree, KD-Tree, and VP-Tree [14, 69, 82] are not as practical or efficient as LSH when dealing with high-dimensional data such as images or audio. R-tree has $O(2^d \log n)$ complexity, where d refers to the index key dimension. The factor 2^d significantly limits its usage in high-dimensional scenarios. Table 2 shows the lookup speed using different data structures when given random high-dimensional vectors as keys. While LSH consistently caps the lookup time at around 0.01 ms, that time for R-tree increases exponentially, to 87 ms, with the number of dimensions.

Limitation of LSH. The standard LSH is statically configured, however, limiting its performance.

Recall that each hashtable in the LSH leverages a set of hash functions $h : R^d \rightarrow \mathbb{N}$. Each hash function maps a vector $\boldsymbol{v}$ to an integer by $h_i(\boldsymbol{v}) = \lfloor \frac{\boldsymbol{a}_i \cdot \boldsymbol{v} + \boldsymbol{b}_i}{r} \rfloor$, where $\boldsymbol{a}_i, \boldsymbol{b}_i$ are random projection vectors and *the parameter r captures the granularity of the buckets, i.e., how well buckets differentiate among dissimilar entries*. The concatenation of the j integers together forms a bucket, $< h_1(\boldsymbol{v}), h_2(\boldsymbol{v}), ..., h_j(\boldsymbol{v}) >$, within the hashtable.

Thus, *configuring r is crucial for LSH performance*. Lookup is both fast and accurate when the hash bucket granularity matches the distribution of the cache keys.

The rightmost part of Figure 3 shows two examples of parameter misconfiguration. The star represents the query input, and should ideally be hashed to a bucket containing all the squares but only squares. When the buckets are too coarse-grained, the *hashing differentiation* is weak. Many dissimilar keys are hashed to the same bucket. Searching through a large bucket is slow, but we can be confident that all relevant entries are in the bucket and the best match can be found. Conversely,fi ne-grained buckets contain few entries each and are quick to search through, but might not contain the best match.

In practice, a major challenge is that the distribution of the input data is unknown and often time-varying. The performance of the standard LSH is thus at the mercy of the data distribution. This necessitates an algorithm to tune the LSH configuration during run time.

Adaptive LSH (A-LSH). In the LSH query complexity expression $O(n^{\rho}\log n)$, we aim to keep the parameter ρ constantly low for optimal lookup performance.

Analytically, ρ is determined by r [21]: $\rho(r) = \log_{p_r} p_1$, $p_r = 1 - 2\Phi(-r/c) - \frac{2}{\sqrt{2\pi}r/c}(1 - e^{-(r^2/2c^2)})$, and p_1 is simply p_r when $r = 1$. $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution, $N \sim (0, 1)$.

c is in fact called the *locality sensitivity factor*. Its ideal value should divide the entire data set into disjoint subsets, such that the variance is small within each subset but large across different subsets. The bucket granularity (r) can then be determined based on the intra-subset variance. Therefore, to obtain the optimal ρ, wefi rst estimate the value of the *data-dependent* parameter c in the previous formula and then optimize the parameter r accordingly.

Since c varies with the current cached key distribution but no assumptions can be made in advance, we approximate c with the statistics of the *key* distribution. Specifically, for each cache key, wefi rstfi nd its k^{th} nearest neighbors, where k is a pre-selected constant (Section 3.3). The distance to this k^{th} neighbor, D_k, measures the radius of the immediate neighborhood cluster of the cached key. Across all cached keys we then have a distribution of D_k. Then, we calculate c as the smaller of $5\times mean(D_k)$ and the 95th percentile within the distribution of D_k. This is an empirical rule we learned from experiments, covering a wide variety of data. Finally, we can tune r to reach the local minimum of the function $\rho(r)$ during the run time, leveraging existing optimization methods such as gradient descent [16].

3.3 Homogenized k Nearest Neighbors

After retrieving several "closest" cached records, we need to determine the reuse output from these records. Intuitively, we want to reuse aggressively as opportunities arise, but also conservatively to ensure the reused result would be identical to a newly computed result. This requires balancing the reuse quality and aggressiveness.

k nearest neighbors (kNN) as a strawman. Selecting a reusable record can be modeled as a data classification problem, so wefi rst consider kNN [10], an algorithm most widely used for this purpose. The algorithmfinds k records closest to the query input, identifies the cluster label associated with each, and then returns the *mode* of the cluster labels through majority voting. When applied to the cached *key-value pairs* for our reuse scenario, step one above is based on matching *keys*, while the "cluster label" is the *value* field of each pair.

The primary advantage of kNN is its non-parametric nature, namely, no data-specific training is needed *a priori*. Despite the simple idea, kNN has been proved to approach the lowest possible error rate when given sufficient data [75]. State-of-the-art improvements, such as weighted kNN [51], assign different weights to the nearest records to further improve the kNN accuracy.

Problem with native kNN. The ideal situation for native kNN is when the k records form a single dominant cluster that truly matches the query key. The *value* associated with this cluster is then unambiguously the correct result for reuse. In practice, however, neither condition is guaranteed. Therefore, native kNN and its variants cannot always ensure accurate reuse. Nor do they give much control over the reuse quality or the aggressiveness. The limitations are manifested in both the input and output processing.

First, existing kNN variants cannot always assess input similarity accurately. The Euclidean distance between high-dimensional vectors (i.e., the cache keys) becomes less informative with increased dimensionality and fails to reflect the similarity in the keys. The *curse of dimensionality* causes the noise in certain dimensions to disproportionally skew the overall distance measurement [10].

Second, a dominant *value* cluster is often absent due to insufficient data or a skewed data distribution, and existing kNN variants provide inadequate *tie-breakers* between multiple clusters. As an example, suppose an input key K_1 is located at the intersection of two clusters, corresponding to the computation outputs V_1 and V_2 respectively. Among the nearest keys of K_1, half of their values are V_1, and the rest are V_2. In this case, either V_1 or V_2 can be valid, and it is impossible to select one correctly without further information.

Consequently, the *perceived input similarity* does not guarantee *output reusability*, and it is hard to gauge the confidence level of correct reuse merely from the cache *keys*. Unfortunately, native kNN and variants make decisions based on the *keys* but not the cached *values*.

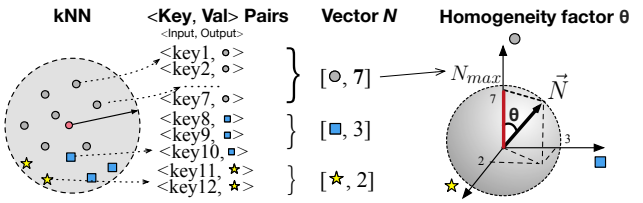


Figure 4. Calculating the homogeneity factor θ.

To address the above limitations, we propose a novel refinement, called *homogenized kNN* (H-kNN). It utilizes the cached *values* to remove outliers and ensure a dominant cluster among the k records initially chosen. This lets us improve and explicitly control the *quality of reuse*.

Homogeneity factor θ. Observe that the kNN performance issues arise from the lack of a suitable mechanism to assess the dominance of the clusters from the k records, hence the correctness of reuse. We therefore define a metric, the *homogeneity factor* (θ), for this purpose.

From the k key-value pairs, wefi rst prepare a frequency vector $\vec{N} = [N_1, N_2, ..., N_k]$, where each element N_i records the frequency of the i^{th} distinct *value*. Then, $\theta(\vec{N}) = N_{max}/||\vec{N}||_2$, where $N_{max} = \max(N_i), \forall\, i \in [1, k]$. Figure 4 shows an example. 12 nearest keys correspond to 3 distinct values. Therefore, we derive $\vec{N} = [7, 3, 2]$, and $\theta = 7/\sqrt{7^2 + 2^2 + 3^2}$.

A geometric interpretation gives an intuition behind θ. Cached records (key-value pairs) with the same *value* form a cluster. Each cluster is mapped to a distinct dimension in $\vec{N}$, and the cluster size mapped to the length of the projection onto that dimension. The homogeneity factor θ is actually the *cosine* distance between $\vec{N}$ and its longest projection. A small *cosine* distance implies the existence of a large dominant *value* cluster, i.e., a high level of homogeneity among the k records selected. In that case, we can be highly confident that this dominant *value* is the correct result for reuse.

This definition of θ applies to discrete output values, which covers most classification scenarios. If, instead, the output values are continuous, θ can simply be defined to be inversely proportional to the *variance* of the k values and normalized to the proper scale.

Homogenized kNN (H-kNN). With the homogeneity factor, we can then set a threshold θ_0 to control the reuse quality. Algorithm 1 describes the operations of H-kNN. The intuition behind our refinement is tofi rst remove outliers from the k records initially chosen and then assess the homogeneity of the remaining records. Reuse proceeds only if there is a dominant cluster. Note that $mean(D_k)$ is the average k^{th} nearest neighbor distance D_k (also used when adapting the LSH parameters in Section 3.2).

The value of θ_0 simultaneously affects the correctness of the returned results and the proportion of non-null query

Algorithm 1: Homogenized kNN

```
// queryKey and k are arguments
Select k nearest neighbors with native kNN, get
  List<record> neighborList;
neighborList.filter {record ⇒ distance(record.key,
  queryKey) < mean(D_k)};
Calculate N⃗ and θ from neighborList;
if θ > θ_0 then
    return value corresponding to N_max;
end
return null;
```

outputs, the latter of which can be interpreted as the aggressiveness of reuse. Therefore, with H-kNN, the *quality of reuse* can be enhanced and explicitly controlled by adjusting θ_0. A lower θ_0 permits more reuse but potentially less accurate results. θ_0 can be set empirically by default (discussed in Section 6.3.1) or dynamically by the application according to its preference for the aggressiveness of reuse.

Bounding accuracy loss. For H-kNN,first note the reuse accuracy is tunable through θ_0. Next, we investigate the error inherent in the H-kNN algorithm.

For an input x, the error probability of reuse can be denoted by $error(reuse) = P_{x\sim C}(reuse(x) \neq compute(x))$, where C is the input distribution. According to the Probably Approximately Correct (PAC) learnability framework [48], for each given constant values of ϵ and δ, the error rate is bounded by ϵ with at least $1-\delta$ probability, when the number of participating samples, n, exceeds the value of the polynomial $p(\frac{1}{\epsilon}, \frac{1}{\delta}, n, dim(f))$. n in our case is the total number of the cached records that are usable by H-kNN. $dim(f)$ is a factor determined by the intrinsic complexity of the learning task. For native kNN, $dim(f)$ quantifies how well the nearest neighbor data points can be unambiguously clustered (i.e., the *VC dimension* of a local sphere constituted by the nearest neighbour data [45]).

In other words, we can bound the potential accuracy loss of *kNN* with a specified confidence level by tuning the bucket granularity of the records stored in A-LSH, i.e., the parameter r mentioned in Section 3.2. As *H-kNN* improves on kNN, the reuse error can be reduced by further factor determined by the intrinsic VC dimensionality of the cached *values*.

3.4 Generality

We emphasize that the approximate reuse algorithms (A-LSH and H-kNN) are applicable to different applications.

Application-agnostic process. A-LSH and H-kNN are agnostic to the application logic because they operate at the granularity of individual functions (e.g., an image recognition method) instead of an application as a whole (e.g., the Google Lens app). If an application successively calls image recognition and speech recognition, say, two separate reuse queries will be issued.

A learning-based application typically executes in three stages: (i) acquiring and preprocessing the input, (ii) invoking relevant machine learning pipelines, and (iii) combining the outputs to generate thefi nal result.

For example, the cognitive assistance app (Section 2.1) might acquire an image from the camera view, run image recognition to identify a landmark label, search for the landmark on the Internet, andfi nally display a selected page to the user. Depending on the input image quality, additional preprocessing can be employed, e.g., illumination correction, noise removal, and segmentation. Similarly, thefi nal output generation steps, e.g., combining outputs from multi-modal learning pipelines, searching for related information and rendering on the screen, would be distinct for each specific case. However, the core machine learning functions, i.e., image recognition, are common across invocations of the app.

Stages (i) and (iii) vary by application and potentially even between different runs of the same application. In contrast, Stage (ii) only varies by the type of learning operations but not the specific application contexts. Operating only on stage (ii) enables A-LSH and H-kNN to be application-agnostic.

Beyond classification. Although we have used classification examples throughout this section, the reuse framework is broadly applicable to different types of machine learning functions. A-LSH is designed for fast and accurate nearest neighbour lookup upon high-dimensional data, and hence is generic to machine learning models. H-kNN can be applied to learning tasks with either discrete output (i.e., classification) or continuous output (i.e., regression, prediction), as explained in Section 3.3. A sufficient condition of H-kNN is local smoothness of the model, which is shown to be satisfied by a majority of the machine learning techniques [25, 32].

4 FoggyCache

The techniques discussed in the previous section are generic to any approximate computation reuse scenarios. In this section, we discuss how to implement these techniques as a service for *cross-device* reuse outlined in Section 2. We target contextual based recognition and inference applications.

Mobile computing paradigms today typically require coordination between the smart devices, nearby edge servers [42], and the remote cloud. An essential component in such systems is an offloading runtime, such as MAUI, Odessa, and Comet [20, 31, 66]. The runtime dynamically partitions the processing pipeline intofi ne-grained tasks and places their execution locally on the device or remotely on a server.

Therefore, we re-design the traditional offloading runtime by incorporating approximate reuse as a service called FoggyCache, interposed between the application and the offloading runtime as an intermediate layer. FoggyCache intercepts the application call to the offloading runtime interface, as shown

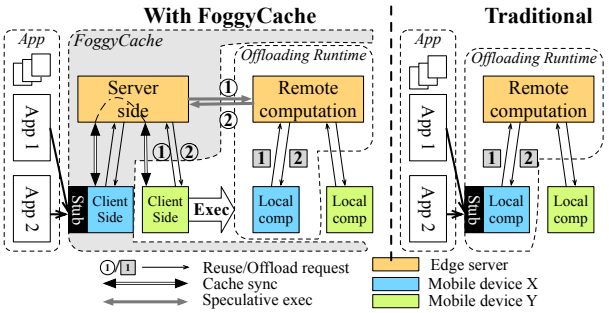


Figure 5. System architecture.

in Figure 5, to invoke approximate reuse regardless of where the computation task is eventually executed.

4.1 System overview

FoggyCache follows a typical client-server architecture. The FoggyCache client can be on a smartphone, tablet, or IoT device. The FoggyCache server is a central point of coordination between the clients. Given the advent of mobile edge computing for low-latency computation offloading, edge servers or cloudlets [71] are ideally suited to deploying the FoggyCache server.

Server. The server-side FoggyCache consists of an A-LSH cache gathering previous computation records (input & output) of all clients, a service daemon handling reuse queries, and a module that handles the client-server coordination.

Client. The client-side FoggyCache consists of an on-device A-LSH cache and a service endpoint which interacts with the server-side cache and the offloading runtime or the applications. The local cache stores a subset of the computation records from the server-side to minimize remote lookup.

Plugging FoggyCache in the offloading runtime. To avoid modifying the application, we retain the native interface between the application and the offloading runtime. The FoggyCache *client* intercepts the offloading call inside the entry point to the offloading runtime.

Take MAUI as an example. Once a method is declared as `remoteable`, its invocation will prompt the standard offloading runtime to schedule the code execution, locally or remotely. With FoggyCache, the method invocationfirst triggers the reuse pipeline before a scheduling decision is made. If any previous results are reusable, these are returned directly to the application without further computation. Otherwise, the normal offloading action resumes to schedule and execute the task. The APIs are detailed in Section 5). The FoggyCache *server* runs in its own process or container, separately from the remote end of the offloading runtime.

Challenge: two-level cache coordination. Both the server-side and client-side caches adopt the *least frequently used* (LFU) policy for cache entry replacement. However, coordination is crucial between the two levels of cache. As new computation requests are initiated from the clients, yet cross-device reuse is supported at the FoggyCache server side. Therefore, how new computation records propagate from the clients to the server and vice versa notably affects the FoggyCache performance. Our solution has two parts, corresponding to the two directions of datafl ow between the client and the server, shown in Figure 5 by the arrows of cache sync and speculative execution.

4.2 Client side cache management

Two mechanisms are needed to synchronize the client side caches with the server side: *client warm-up* and *cache miss handling*. The former is needed when a client appears in the vicinity of the server for thefi rst time to boot-strap the service. The latter is triggered when no locally cached outputs could be reused.

Client cache warm-up. Intuitively, the client cache should receive broadly distributed entries to jump start the reuse service and maximize the probability of a random reuse query being matched with a reusable output from the cache.

Without relying on any assumptions on the input data distribution, we adopt *stratified sampling* to generate a subset of the server cache. The size of the subset is determined by the client or follows a default value. Algorithm 2 details the operations. The key idea is tofi rst select as many *types* of cached *keys* with popular cached *values* as possible, where the *popularity* of a cached value is estimated based on the number of cache keys mapped to this value. This ensures a broad coverage of the records in the subset so that our approximate reuse algorithms could proceed in most cases. If space remains in the client cache after thisfi rst sampling pass, the algorithm then proportionally samples from the rest of the entire server cache. This algorithm achieves a dynamic trade-off between the subset coverage, the distribution of the cache keys, and the limited client cache space.

Note that existing data-dependent prefetching techniques [62, 63] are orthogonal to our design. While we aim for reasonable performance *without prior knowledge*, other prefetching techniques could be adopted instead if *prior knowledge about the input data is known*.

Cache miss handling. Cache miss handling is on the critical path of the application, and therefore the processing logic and the data transmitted should be lightweight and minimal.

When the FoggyCache client does not carry reusable outputs, it sends a request to the server including the query input and the *homogeneity threshold* θ_0. Different clients could therefore customize the tradeoff between time saving and reuse accuracy by querying the same server cache with different thresholds θ_0.

The FoggyCache server executes the query and returns the reused output along with k nearest records, the minimum needed to carry out the H-kNN algorithm. This way, it reduces a potential cache miss from a similar, future query

Algorithm 2: Initial cache warm-up algorithm

```
// Subset size s and num of nearest neighbors
   k are inputs, subset is the output
Initialize empty subset[s];
Create inverted index Idx: {Value ↦ List<Entry> lst}
  from the cache;
Sort the Value from large to small in Idx w.r.t. lst.size();
Store the sorted Values into a List<Value> vlist;
while vlist is not empty && subset is not full do
    Value v = vlist.get(0);
    List<Entry> elist = Idx.get(v);
    Sample min(k, elist.size(), subset.spaceLeft()) entries
      from elist and append them to subset;
    vlist.remove(v);
end
if vlist is empty then
    Proportionally sample entries from the cache tofill
      up the subset;
end
```

on the client device. Meanwhile, k is small enough to avoid incurring non-negligible communication overhead.

4.3 Server side cache updates

From the server perspective, it is desirable to collect newly generated computation records from the clients in a timely fashion for cross-device reuse. Intuitively, each FoggyCache client can batch updates to the server periodically. However, the cache entries might reach the server too slowly and unreliably this way, especially in the face of client mobility or unstable network connectivity. Moreover, not all computation records are created equal. For instance, a computation record with few nearest neighbour records stored in the FoggyCache server could potentially benefit all clients that submit reuse queries with similar inputs, and hence should be synchronized to the server as soon as possible. However, only the FoggyCache server knows such information. Therefore, we devise a speculative execution mechanism on the FoggyCache server to speed up its updates proactively.

Speculative computation. Once a reuse query comes, the server additionally estimates the *importance* (i.e., the probability of future reuse) of the computation task that corresponds to the query. Based on this probability, the server decides whether to speculatively execute the task and add the input-output record to the cache for future reuse queries.

Although prediction-based speculative execution algorithms are widely used [60, 74], they are not directly applicable. Due to the approximate nature, the *importance* of a computation record is no longer solely decided by the access statistics about the record itself.

Instead, the likelihood of a computation record being reused in the future is jointly determined by three factors: the average access frequency (F_k), the average distance from the reuse query ($Dist_k$), and the homogeneity factor (θ_k) among the k nearest neighbors of the query input. The FoggyCache server also maintains the average access frequency F_{avg}, the average distance to k nearest neighbor $Dist_{avg}$ among all cached records, and the default homogeneity threshold θ_0. We then calculate $P_f = \min(F_{avg}/F_k, 1)$, $P_d = \min(Dist_{avg}/Dist_k, 1)$, and $P_\theta = \min(\theta_k/\theta_0, 1)$, as the corresponding normalized factors ranging in (0, 1] so that they can be used as probabilities. Then, $importance = 1 - P_f \cdot P_d \cdot P_\theta$. The multiplication captures the independence between these three factors. The FoggyCache server will then invoke speculative computation for this query with a probability equals to *importance* . The intuition behind the *importance* value is that wefi rst take access frequency F_k as a baseline estimate, and then further consider the approximate nature, where the input distribution ($Disk_k$) and the output distribution (θ_k) both play an important role in determining the reused output (Section 3).

Note that the decision to proceed with speculative execution does not consider the load on the edge server. Basically, we decide *whether* to speculatively compute a record based on its importance, but we let the task scheduler on the edge server to decide *when* to execute the speculation task. For instance, the task can be separately assigned a low priority to avoid it contending with latency-sensitive tasks.

4.4 Additional consideration

Incentives: Various approaches [18, 26, 59] have been proposed to incentivize participation in decentralized systems. FoggyCache follows the "give and take" approach to incentives, similar to the proposal in [26]. Each FoggyCache client is allocated free credits at the beginning, while additional credits are given *proportional* to the number of computation outputs it contributes. The credits are used to query reusable computation from the server. The exact numerical value of the proportion parameters vary based on the global balance of queries and contributions under each specific scenario.

Security. The main security concern for FoggyCache arises from malicious devices polluting the cache with false computation outputs. To address this, we can incorporate existing object-based reputation system (e.g., Credence [80]) in FoggyCache with negligible additional overhead. Each computation record is additionally labelled with an anonymous identity of the contributing client. Clients implicitly vote on cached records while running the reuse query. Specifically, if a cached record is selected by Step 2 of the H-kNN but its output is not chosen in the end, this constitutes a negative vote. Conversely, a successful reuse is a positive vote.

Privacy. Good enough privacy could be achieved by anonymizing participating devices when reporting data to the server. Since FoggyCache targets locality-based scenarios, the raw input of the approximate reuse is mostly local contextual

information. Such information is meant to be collected by all nearby entities and hence public by nature. Location privacy is less of a concern here. Moreover, the FoggyCache client does not have to store and operate on raw input data. This means that different applications or vendors can employ their custom encryption schemes to protect the raw data without affecting cross-device reuse, as long as they feed the feature vectors extracted to FoggyCache.

5 Implementation

5.1 Architecture

We implement FoggyCache following a typical client-server model. A two-level cache structure that spans the edge server as well as the local device serves as our storage layer. The communication layer builds on the Akka [2] framework.

Cache layout. The two-level storage adopts the same layout. The highest level of each cache structure is a Java `HashMap`, which maps a function name (`String`) to an in-memory key-value store, where an A-LSH is generated from the key region among computation records of this function collected from all the clients. Additionally, the server side cache system includes utility functions to serialize and deserialize its data partially to disk.

Concurrency. FoggyCache is built using the Akka toolkit, which adopts the actor model [9] to handle concurrent interactions between system components. Each function module is implemented in a separate class extending the Akka `AbstractActor` class. Concurrency is managed implicitly by the Akka framework via message passing. We further leverage the Akka cluster module to provide a fault-tolerant, decentralized membership service.

5.2 APIs and patches

FoggyCache APIs. As much as possible, FoggyCache aims to make the processing logic transparent to the offloading runtime and applications. Therefore, three intuitive APIs are exposed: `ConfigFunc(func_name, config)`, `QueryCompOutput(func_name, input, params)`, and `AddNewComp(func_name, input, output)`. Thefirst specifies reuse configurations for each native function (e.g., serialization, feature extraction, and vector distance calculation). The latter two trigger reuse queries and feed the native processing outputs back to FoggyCache.

Application or library patches. To interact with FoggyCache, short patches should be applied to the offloading runtime, or the application code when no runtime is used. No more than 10 lines of code is needed to wrap around the native pipeline. `QueryCompOutput` and `AddNewComp` are added to the native code within a conditional statement to determine whether to invoke the native processing pipeline. `ConfigFunc` enables on-demand customizations.

Table 3. Data correlation in different settings.

Setting	Avg norm distance
ImageNet (same synset)	1.00 +/- 0.15
Video (10 frames apart)	0.31 +/- 0.04
Video (30 frames apart)	0.53 +/- 0.27

6 Evaluation

6.1 General setup

Application benchmarks. Following the motivating examples in Section 2, we build three stripped-down versions of real applications as benchmarks, two for image recognition (plant and landmark detection) and one for speaker identification. These are implemented in Java, using the DL4J [3], OpenCV [17], and Sphinx [7] libraries. The workload settings follow those in related papers [30, 47], using the same pre-trained neural network models that are widely adopted by real applications. Compared to the real applications, our benchmarks skip supporting functionalities such as the user interface, since they can interfere with the timing and energy measurements of the core computation modules. Our benchmarks can also be instrumented easily for various measurements, which is difficult with proprietary applications.

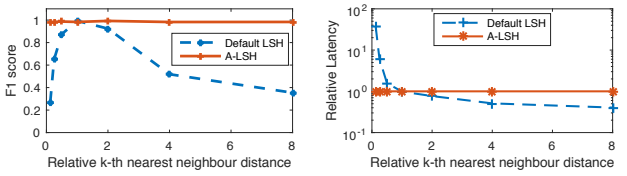
Datasets. We use two standard image datasets, ImageNet [67] and Oxford Buildings [64], an audio dataset, TIMIT acoustics [27], and several real video feeds.

The ImageNet plant subset includes over 4000 types of labeled plant images, taken from different viewpoints under various lighting conditions. The Oxford Buildings dataset consists of 5000 images of 11 specific landmarks in Oxford, hand-picked from Flickr. The TIMIT acoustic dataset [27] contains broadband recordings of 630 speakers of eight major dialects of English, and we use it for speaker identification. For end-to-end performance evaluation, we also use several 10-minute real video feeds, taken on a university campus and in a grocery store, multiple times at either location.

Table 3 compares the average feature vector distance between two images from the same syntax set in ImageNet and two frames from a video feed. The distance is significantly larger (by more than 50%) for ImageNet than for successive video frames, because no *spatio-temporal correlation* exists between images in ImageNet.

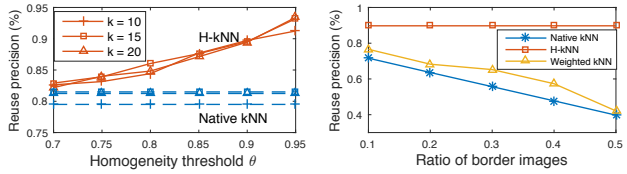
Therefore, we mainly use the standard image and audio datasets in our evaluation. Although they appear less realistic than real audio or video feeds, they present more challenging cases for computation reuse and help us gauge the lower-bound performance of FoggyCache.

Hardware setup. With a 64-bit NVIDIA Tegra K1 processor, Google Nexus 9 is one of the most powerful commodity Android mobile devices. Thus, we use the tablet (running Android OS 7.1) as the client side device to assess the potential benefit from saving computation with FoggyCache. The FoggyCache server runs on a Ubuntu (14.04) Linux desktop



(a) F1 score (b) Latency

Figure 6. Lookup quality and latency for the default LSH and A-LSH.



(a) Effects of θ_0 (b) Effects of data distribution

Figure 7. Reuse precision of H-kNN and alternatives.

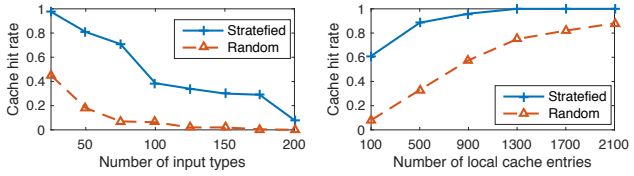
server with a quad-core 2.3 GHz Intel Xeon CPU and 16 GB of memory.

6.2 Microbenchmarks

A-LSH performance. Wefi rst select a subset from ImageNet, optimize the parameter r for the default LSH, and calculate the average k^{th} nearest neighbor distance ($mean(D_k)$ in Section 3.2). Recall that this distance captures the density of the data in the LSH (a large distance indicates a low density and vice versa). Then, we select other subsets of images where their average k^{th} nearest distances range from 1/8 of the D_k to 8×. These subsets serve as the input to the default LSH and A-LSH. k is set to the default value 10. The lookup quality is measured by the *F1 score*, the harmonic mean of *precision* (the correct rate of the results) and *recall* (the percentage of the correct results found), ranging from 0 (the worst) to 1 (the optimal).

Figure 6(a) shows that the lookup quality of the default LSHfl uctuates dramatically given different data densities, whereas A-LSH consistently maintains an F1 score over 0.98. The default LSH only achieves a high lookup quality when the data distribution matches the pre-determined value of r. Figure 6(b) further shows that the lookup time for A-LSH remains constant. However, there is no guarantee for the default LSH, especially when the data are densely stored and thus highly clustered into the same few hash buckets. Note that although LSH appears to incur a lower lookup time for sparsely populated data, the corresponding lookup quality is low. Together thefi gures show that A-LSH accurately adapts the parameters to the dynamics of the input data distribution, and *consistently achieves a near-optimal balance between the lookup quality and speed.*

H-kNN performance. We compare H-kNN with naive kNN and a state-of-the-art variant, weighted kNN [51]. The performance metric is the reuse *precision*, which is upper-bounded by 100%.



(a) cache size 1000 (b) 100 types of inputs

Figure 8. Client cache hit rates and server cache sampling strategies.

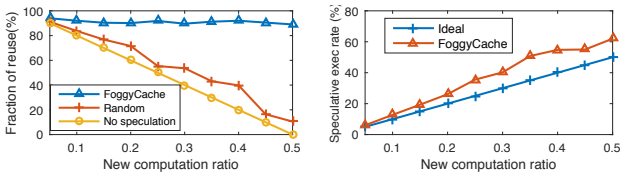
First, we select 1100 images from 4 types of syntax sets in ImageNet. 1000 of them are fed into the cache, and the other 100 images as inputs for H-kNN k and θ_0 (the homogeneity threshold) values vary. The solid and dashed lines in Figure 7(a) represent the H-kNN and native kNN performance, respectively. H-kNN outperforms native kNN by an increasing margin as θ_0 increases, which confirms that (i) the homogenization process improves the reuse accuracy, and (ii) the level of accuracy is indeed tunable through the parameter θ_0. This means that *applications can customize the level of reuse based on desirable accuracy guarantees.* The value of k makes little difference, however. Based on the results, we set k = 10 and θ_0 = 0.9 throughout this section. More detailed tradeoff is shown in Figure 10(b).

Second, we investigate how H-kNN copes with two intersecting clusters (the example against native kNN in Section 3.3), by adjusting the proportion of cache keys at the intersection of two clusters. Figure 7(b) indicates that H-kNN maintains a consistent and high reuse precision regardless of the key distribution. Unfortunately, both native and weighted kNN suffer, as predicted in Section 3.3, with the reuse precision dropping by 40%.

Client cache warm-up. We next evaluate the benefit of stratified sampling for client cache warm-up (Section 4.2), and compare that to randomly sampling server cache entries.

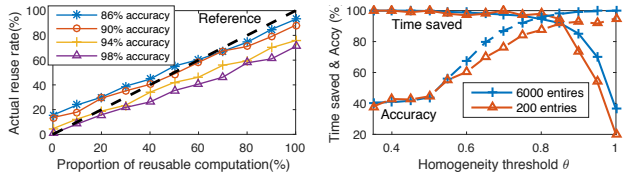
We generate different key-value pairs from ImageNet data to store in the FoggyCache server cache and also for reuse queries. Then, we bootstrap the client cache with stratified sampling and random sampling (as the baseline) respectively. The performance of the algorithms is shown in terms of the client cache hit rate.

First, we set the client cache size to 1000 entries, change the number of syntax sets of images at the server, and observe the cache hit rates. We make two observations from Figure 8(a). (i) When fewer than 100 types of images are cached at the server, stratefied sampling achieves over 50% cache hit rate, which confirms that popular images in ImageNet are adequately prioritized. (ii) When more types of images are cached at the server, the client cache hit rate from random sampling drops to nearly zero, whereas stratified sampling still manages over 25% hit rate, showing better type coverage in the latter.



(a) Speculation performance (b) Speculation efficiency

Figure 9. Performance comparison of speculative execution in FoggyCache and alternatives.



(a) Reuse opportunities captured (b) Reuse vs accuracy tradeoff

Figure 10. Tradeoff between captured reuse opportunities and computation accuracy.

Then, we select 100 types of images from the server cache and vary the client cache size. Figure 8(b) shows both strategies achieve 80% hit rate, but stratefied sampling requires only a quarter of the cache space needed by random sampling.

Speculative server cache updates. Finally, we gauge the benefit of incorporating speculative computation (Section 4.3) in FoggyCache. We select a subset of ImageNet dataset to create multi-device reuse query streams, where the fraction of "new" computation (no reusable results exist at all) ranges from 5% to 50%. We compare our speculative execution algorithm with two alternatives, *random* and *no speculation.Random* means invoking speculative execution with the same probability as in FoggyCache, but selecting inputs randomly. The ideal reuse proportion is 100%.

Figure 9(a) illustrates that FoggyCache consistently caches in around 90% of the reuse opportunities, whereas *random* and *no speculation* cannot keep up as the fraction of "new" computation increases, because FoggyCache accurately predicts the *importance* of a computation record for future reuse and pre-emptively generates that record before the actual reuse request. Figure 9(b) compares the fraction of computation that is speculatively executed in the ideal case (each speculatively generated record is visited later) and in FoggyCache, and our algorithm only triggers 10% unnecessary computation at most compared to the ideal case.

6.3 FoggyCache performance

6.3.1 Tradeoff between reuse and accuracy

Accuracy. We run object recognition using ResNet50 [37] on selected ImageNet images to assess the tradeoff between the aggressiveness of reuse and the accuracy.

First, we quantify how well FoggyCache recognizes reuse opportunities when the fraction of reusable queries in the whole query stream varies. The dashed line in Figure 10(a) shows the ideal case and serves as a reference. Any points above indicate false negatives (missed reuse), while points below the line indicate false positives (inaccurate reuse).

FoggyCache consistently captures the reuse opportunities in all data combinations while maintaining high accuracy. Both the false positive and false negative rates are below 10% (the 0% and 100% reuse points) while the reuse accuracy exceeds 90%. Even if we reuse conservatively to ensure a 98% accuracy, we only miss fewer than 30% of all reuse chances.

Second, we examine the trend of the total computation time saved and the relative accuracy (compared to native recognition accuracy), both as the homogeneity threshold θ_0 varies. We run the experiments for various caching levels, ranging from 200 to 6000 cached entries. For legibility only the lines for 200 and 6000 entries are plotted. The other lines fall between these two.

The dashed and solid lines in Figure 10(b) plot the relative accuracy and the time saved respectively. We can see that setting θ_0 to between 0.8 to 0.95 would ensure both higher than 90% accuracy and less than 20% loss of the reusable opportunity. This confirms that FoggyCache achieves a decent balance between accuracy and computation time reduction.

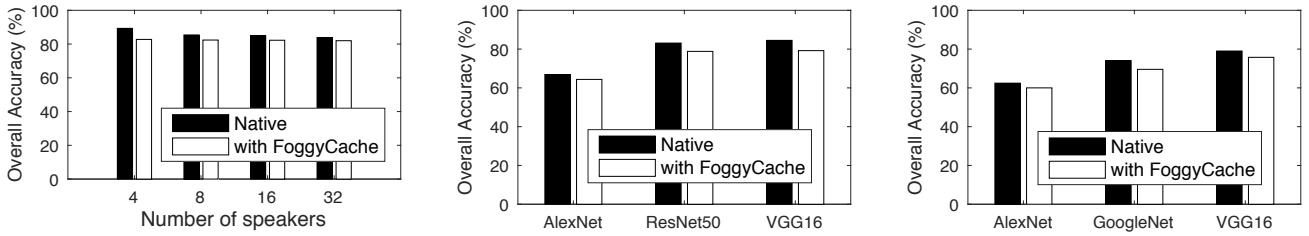
User experience. We conduct an informal user survey among students on our campus to gauge how approximate reuse affects user experience. In the context of the cognitive assistance application, students are asked whether they are satisfied with different combinations of the percentage accuracy loss and the reuse benefits (in terms of percentage reduction of battery consumption and latency), with data points taken from Figure 10(b). From 100 completed questionnaires, 92 are satisfied with the user experience when the accuracy loss is under 5%, and 80 satisfied when the accuracy loss is under 10%. FoggyCache performs well for both cases. For more accuracy-sensitive applications, such as autonomous driving and medical pill recognition, the accuracy of FoggyCache can be tuned by carefully selecting the value of θ_0 and the number of cached records.

6.3.2 End-to-end system performance

We investigate the end-to-end performance of FoggyCache using the three aforementioned application benchmarks. We separately consider two modes of execution for mobile applications, local processing on the mobile device and edge offloading. The real-time decision made by the offloading runtime between the two modes is orthogonal to the FoggyCache behavior. The performance metrics are *latency*, *energy consumption*, and *accuracy*. The *latency* is measured end-to-end from the arrival of a request to its completion. The *accuracy* is defined as the percentage of correct results. The *energy consumption* is calculated based on the real-time battery status collected with the Android debugging API, `adb dumpsys batterystats`.

Table 4. End-to-end FoggyCache performance.

Workload			Latency (ms)				Energy (mJ)	
Application	Description		Offl. (w/o)	Offl. (w/)	Without	With	Without	With
Speaker Identification	Number of speakers	4	28.1	***8.4***	13.1	***4.2***	30.4	***9.8***
		8	28.1	11.8	13.1	5.5	30.4	13.3
		16	28.1	12.1	13.1	5.9	30.4	13.7
		32	28.1	13.2	13.1	6.4	30.4	15.0
Landmark Detection	Types of neural network	AlexNet [46]	24.6	16.3	37.1	19.5	365.9	39.5
		ResNet50 [37]	32.8	17.7	102.4	27.9	1315	110.7
		VGG16 [73]	53.8	***21.4***	269.6	***57.3***	3132	***246.9***
	Video feed (campus) w/ VGG16		53.8	***12.0***	269.6	***25.4***	3132	***114.2***
Plant Recognition	Types of neural network	AlexNet	24.6	16.6	37.1	21.4	316.8	113.9
		GoogleNet [76]	29.2	17.9	65.3	32.2	817.4	236.8
		VGG16	53.8	***27.9***	269.6	***99.8***	3132	***901.4***
	Video feed (grocery) w/ VGG16		53.8	***16.5***	269.6	***30.8***	3132	***131.1***



(a) Speech identification (b) Landmark recognition (c) Plant recognition

Figure 11. The accuracy of the processing pipeline with and without FoggyCache.

Experiment settings. We use Nexus 9 tablets as the FoggyCache clients, configured with a 15 MB local cache size. The FoggyCache server is deployed on a Linux machine which also serves as the edge offloading destination. The network latency between the clients and the server is around 20 ms, a typical value for the edge setting [41]. We also tried lower latencies but they only made FoggyCache perform better.

For *Speaker identification*, we randomly select 3200 speech segments from 4, 8, 16, and 32 speakers respectively from the TIMIT dataset, add different ambient noise, and extract the PLP feature vectors We store the computation records in the server cache, and populate 10% of them to the tablet for client cache warmup. Another 200 speech segments are selected from TIMIT and preprocessed the same way to serve as the test inputs. The core computation of the workload follows the same setting as for DeepEar [47].

For *landmark detection* and *plant recognition*, we take 5000 images each from the Oxford dataset and ImageNet and both the campus and grocery store video feeds. The standard datasets exhibit no *spatio-temporal correlation* between successive inputs, while the real video feeds contain common *imperfections*, e.g., motion induced or out-of-focus blur. We extract feature vectors, warm up the client cache with 10% of the data, and process another 1000 images as test inputs. Fourfi ne-tuned neural network models (AlexNet, GoogleNet, ResNet50, and VGG16) are used to evaluate the performance.

Performance. Table 4 records the performance in all the experiments. "With" and "Without" refer to whether FoggyCache is enabled, and "Offl." refers to cases where the actual computation happens at the edge server instead of the local device. The numbers in bold highlight the most remarkable performance of FoggyCache. FoggyCache achieves a 50-70% latency reduction for both local processing and edge offloading for the standard image datasets. When using the real video feeds, the processing latency could be reduced by 88%. The energy consumption is only measured for local processing. FoggyCache reduces the native energy consumption by a factor of 3 for the standard datasets and 20 for the video feeds. Figure 11 shows that FoggyCache caps the overall accuracy penalty under 5% while achieving good performance. This confirms that A-LSH and H-kNN can ensure the reuse quality regardless of the specific settings. The accuracy penalty for the video feeds is constantly under 1%, hard to tell from the bars and thus not shown in thefi gure.

To sum up, FoggyCache effectively reduces the latency and energy consumption (for on-device processing) of the native processing pipelines, and the benefit is more pronounced when the native logic is more resource intensive.

6.3.3 Large-scale experiment

Finally, we run the landmark detection benchmark and examine how the number of devices affects the overall computation reuse opportunities.

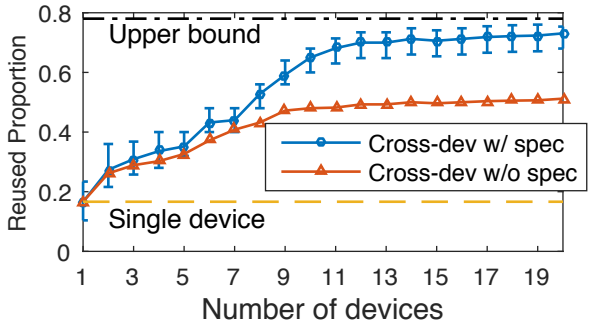


Figure 12. Single- or cross-device reuse achieved with FoggyCache, with or without speculation.

Each client device is supplied with 300 input images about 5 landmarks, randomly selected from the Oxford dataset. This corresponds to feeding to the application a 10-second video at 30 fps with no inter-frame correlation. The server and clients caches are configured to be the same sizes as in Section 6.3.2 but remain empty before the start, so that the caches evolve solely with the reuse needs on all client devices. We start with a single FoggyCache client, and then increase the number of concurrent clients in successive runs.

Figure 12 compares the reuse opportunities captured in several scenarios: reuse within the same device only (the "single-device reuse" line), cross-device reuse with or without speculative execution, and the theoretic upper bound which quantifies the intrinsic reuse opportunity within the input dataset. The error-bars are obtained from 10 runs. As the number of devices in the system increases, the percentage of successful reuse also climbs quickly. Once we have more than 10 devices in the system, the reuse proportion stays above 70%. Additional devices provide marginal benefit, given the upper bound is around 80%. Thefi gure also shows that FoggyCache outperforms intra-device reuse by more than 55% and FoggyCache without speculation by 25%.

7 Related Work

We are aware of little existing work exploring approximate computation reuse *algorithms* in mobile scenarios. Further, the FoggyCache *system* incorporates these approximate algorithms into the existing execution runtime on mobile devices. We discuss a few approaches closest to ours.

Precise redundancy elimination. Redundancy elimination (RE) is widely employed, e.g., in mobile applications, data analytics [33, 44, 56, 65], networking [11, 70], and storage systems [24]. However, existing schemes involve *exact matching* while FoggyCache handles *fuzzy redundancy*.

Starfish [52], MCDNN [36], and Flashback [15] all include caching while accelerating computation-intensive mobile applications. However, they either involve exact matching with cache entries or consider only low-dimensional input values within a pre-defined set. In contrast, FoggyCache handles fuzzy input matching without prior knowledge of the data distribution.

UNIC [77] targets security aspects of RE, which are orthogonal to our design and can be combined with FoggyCache.

Approximate techniques. Approximate *caching* techniques such as Doppelgänger [57] and Image set compression [72] leverage similarity between data pieces to reduce storage overhead. Approximate *computing* techniques such as ApproxHadoop [29] and Paraprox [68] selectively skip inputs and tasks to reduce computation complexity with tolerable errors. FoggyCache adopts similar insights such as exploiting similarity and suppressing error propagation but approximates *repeated computation* using different techniques.

Cachier[23] and Potluck[34] are the closest to FoggyCache. Cachier alludes to the notion of approximate reuse but focuses on cache entry optimization, assuming certain query patterns. Our own prior work, Potluck, experiments with avoiding duplicated image recognition and augmented reality rendering for single device. In contrast, FoggyCache is more general, achieving high quality reuse and tunable performance, without assumptions about the workload.

Cross-device collaboration. Collaborative sensing and inference systems such as CoMon [50] and Darwin [58] revolve around multi-device coordination in the same context. However, unlike FoggyCache eliminating fuzzy redundancy between devices, these cross-device collaboration works focus on partitioning a big job into correlated or independent subtasks, distributing them among the devices, and then collecting the individually results.

8 Conclusion

In this paper, we argue for *cross-device approximate computation reuse* for emerging mobile scenarios, where the same application is often run on multiple nearby devices processing similar contextual inputs. Approximate reuse can simultaneously achieve low latency and accurate results, and is a promising optimization technique.

We design techniques, adaptive locality sensitive hashing (A-LSH) and homogenized k nearest neighbors (H-kNN), to address practical challenges to achieve generic approximate computation reuse. We then build FoggyCache, which extends the mobile offloading runtime to provide approximate reuse as a service for mobile edge computing. Evaluation shows that, when given 95% accuracy target, FoggyCache consistently harnesses over 90% of all reuse opportunities, which translates to reduced computation latency and energy consumption by a factor of 3 to 10. FoggyCache provides tuning mechanisms to further improve the accuracy.

While FoggyCache is optimized for multi-device mobile and edge scenarios, our reuse techniques A-LSH and H-kNN are generic and have broader applicability. We will investigate other approximate reuse paradigms in future work.

References

[1] Amazon Alexa: afi rst look at reachfi gures. https://omr.com/en/

amazon-alexa-skill-marketing/.
[2] Build powerful reactive, concurrent, and distributed applications more easily. https://akka.io/.
[3] Deep learning for Java. https://deeplearning4j.org/.
[4] Google Street View Image API. https://developers.google.com/maps/documentation/streetview/intro.
[5] Let smartphone cameras understand what they see. https://techcrunch.com/2017/05/17/google-lens/.
[6] Let smartphone cameras understand what they see. https://itunes.apple.com/us/app/ikea-place/.
[7] Open source speech recognition toolkit. https://cmusphinx.github.io/.
[8] Smart home, seamless life: Unlocking a culture of convenience, January 2017.
[9] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
[10] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.
[11] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.
[12] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.
[13] M. V. Barbera, A. Epasto, A. Mei, S. Kosta, V. C. Perta, and J. Stefa. CRAWDAD dataset sapienza/probe-requests (v. 2013-09-10). Downloaded from https://crawdad.org/sapienza/probe-requests/20130910, Sept. 2013.
[14] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.
[15] K. Boos, D. Chu, and E. Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 291–304. ACM, 2016.
[16] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
[17] G. Bradski. The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123, 2000.
[18] L. Buttyan and J.-P. Hubaux. *Security and cooperation in wireless networks: thwarting malicious and selfish behavior in the age of ubiquitous computing*. Cambridge University Press, 2007.
[19] 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.
[20] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.
[21] 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*, pages 253–262. ACM, 2004.
[22] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 25–25. USENIX Association, 2012.
[23] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan. Cachier: Edge-caching for recognition applications. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 276–286. IEEE, 2017.
[24] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. *White Paper*, 223310, 2011.
[25] D. Duvenaud, O. Rippel, R. Adams, and Z. Ghahramani. Avoiding pathologies in very deep networks. In *Artificial Intelligence and Statistics*, pages 202–210, 2014.
[26] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Micro-blog: sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 174–186. ACM, 2008.
[27] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, N. L. Dahlgren, and V. Zue. Timit acoustic-phonetic continuous speech corpus. *Linguistic data consortium*, 10(5):0, 1993.
[28] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.
[29] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.
[30] A. Gordo, J. Almazán, J. Revaud, and D. Larlus. Deep image retrieval: Learning global representations for image search. In *European Conference on Computer Vision*, pages 241–257. Springer, 2016.
[31] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *OSDI*, volume 12, pages 93–106, 2012.
[32] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.
[33] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8, 2010.
[34] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284. ACM, 2018.
[35] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.
[36] 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*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.
[37] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
[38] H. Hermansky. Perceptual linear predictive (plp) analysis of speech. *the Journal of the Acoustical Society of America*, 87(4):1738–1752, 1990.
[39] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
[40] R. Hu and J. Collomosse. A performance evaluation of gradientfield hog descriptor for sketch based image retrieval. *Computer Vision and Image Understanding*, 117(7):790–806, 2013.
[41] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 5. ACM, 2016.
[42] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11(11):1–16, 2015.
[43] R. Huitl, G. Schroth, S. Hilsenbeck, F. Schweiger, and E. Steinbach. TUMindoor: An extensive image and point cloud dataset for visual indoor localization and mapping. In *Proc. of the International Conference on Image Processing*, Orlando, FL, USA, Sept. 2012. Dataset available at http://navvis.de/dataset.
[44] K. Kannan, S. Bhattacharya, K. Raj, M. Murugan, and D. Voigt. Seesaw-similarity exploiting storage for accelerating analytics workflows. In *HotStorage*, 2016.

[45] S. Kpotufe. k-nn regression adapts to local intrinsic dimension. In *Advances in Neural Information Processing Systems*, pages 729–737, 2011.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[47] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[48] F. Laviolette and M. Marchand. Pac-bayes risk bounds for stochastic averages and majority votes of sample-compressed classifiers. *Journal of Machine Learning Research*, 8(Jul):1461–1487, 2007.

[49] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.

[50] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song. Comon: Cooperative ambience monitoring platform with continuity and benefit awareness. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2012.

[51] Y. Li and B. Cheng. An improved k-nearest neighbor algorithm and its application to high resolution remote sensing image classification. In *Geoinformatics, 2009 17th International Conference on*, pages 1–4. Ieee, 2009.

[52] R. LiKamWa and L. Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 213–226. ACM, 2015.

[53] B. Logan et al. Mel frequency cepstral coefficients for music modeling. In *ISMIR*, 2000.

[54] 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.

[55] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 7–ff. ACM, 1997.

[56] F. D. McSherry, R. Isaacs, M. A. Isard, and D. G. Murray. Differential dataflow, Oct. 20 2015. US Patent 9,165,035.

[57] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61. ACM, 2015.

[58] E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell. Darwin phones: the evolution of sensing and inference on mobile phones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 5–20. ACM, 2010.

[59] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336. ACM, 2008.

[60] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributedfi le system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 191–205. ACM, 2005.

[61] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.

[62] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[63] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 275–284. ACM, 2013.

[64] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[65] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud*, 2009.

[66] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[67] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[68] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 35–50. ACM, 2014.

[69] H. Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.

[70] S. Sanadhya, R. Sivakumar, K.-H. Kim, P. Congdon, S. Lakshmanan, and J. P. Singh. Asymmetric caching: improved network deduplication for mobile devices. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 161–172. ACM, 2012.

[71] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[72] Z. Shi, X. Sun, and F. Wu. Feature-based image set compression. In *Multimedia and Expo (ICME), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.

[73] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[74] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[75] C. J. Stone. Consistent nonparametric regression. *The annals of statistics*, pages 595–620, 1977.

[76] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[77] Y. Tang and J. Yang. Secure deduplication of general computations. In *USENIX Annual Technical Conference*, pages 319–331, 2015.

[78] H. Verkasalo. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing*, 13(5):331–342, 2009.

[79] S. Vijayarangan, P. Sodhi, P. Kini, J. Bourne, S. Du, H. Sun, B. Poczos, D. Apostolopoulos, and D. Wettergreen. High-throughput robotic phenotyping of energy sorghum crops. *Field and Service Robotics. Springer*, 2017.

[80] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peerfi lesharing. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[81] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[82] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.

[83] M. Zhao, D. Wang, Z. Zhang, and X. Zhang. Music removal by convolutional denoising autoencoder in speech recognition. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2015 Asia-Pacific*, pages 338–341. IEEE, 2015.