Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications

Peizhen Guo
Yale University
peizhen.guo@yale.edu

Wenjun Hu
Yale University
wenjun.hu@yale.edu

Abstract

Emerging mobile applications, such as cognitive assistance and augmented reality (AR) based gaming, are increasingly computation-intensive and latency-sensitive, while running on resource-constrained devices. The standard approaches to addressing these involve either offloading to a cloud(let) or local system optimizations to speed up the computation, often trading off computation quality for low latency.

Instead, we observe that these applications often operate on similar input data from the camera feed and share common processing components, both within the same (type of) applications and across different ones. Therefore, deduplicating processing across applications could deliver the best of both worlds.

In this paper, we present Potluck, to achieve *approximate deduplication*. At the core of the system is a cache service that stores and shares processing results between applications and a set of algorithms to process the input data to maximize deduplication opportunities. This is implemented as a background service on Android. Extensive evaluation shows that Potluck can reduce the processing latency for our AR and vision workloads by a factor of 2.5 to 10.

ACM Reference Format:
Peizhen Guo and Wenjun Hu. 2018. Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/http://dx.doi.org/10.1145/3173162.3173185

ACM ISBN ISBN 978-1-4503-4911-6/18/03...$15.00
https://doi.org/http://dx.doi.org/10.1145/3173162.3173185

1 Introduction

Many emerging mobile applications increasingly interact with the environment, process large amounts of sensory input, and assist the mobile user with a range of tasks. For example, a personal assistance application can "see" the environment and generate alerts or audio information for visually-impaired users [8]. A driving assistance application [44] can render 3D scenes overlaid on the physical environment to help the driver to visualize the surroundings beyond the immediate views. These applications are usually computation-intensive and latency-sensitive, while running on resource-constrained devices.

The standard approaches to resolving these challenges involve either offloading to a cloud(let) [18, 21, 40, 43] or local system optimizations to speed up the computation [15, 32], often trading off computation quality for low latency.

Instead, we observe that these applications often operate on similar, correlated input data with and share common processing components, both within the same (type of) applications and across different ones. While the input data are rarely the same, they share temporal, spatial, or semantic correlation due to the scene change behavior or the requirements of the applications. Moreover, a vast majority of these applications exhibit a unique computation feature in common: correlated and similar input values often yield the same processing results. Being lifestyle applications, it is highly probable for these applications to be installed on the same device [34]. This suggests that deduplicating processing across applications and inputs could deliver the best of both worlds, i.e., achieving good performance within the resource constraints. Section 2 discusses these opportunities in detail. Deduplication is orthogonal to both offloading and local system optimizations, and can be combined with either for further optimization.

While recent works involve sharing computation, we argue for more generic deduplication. Specifically, StarFish [33] shares some intermediate results at the library level for computer vision applications, MCDNN [23] enables sharing results from some layers of the deep neural network for different applications operating on top, and FlashBack [14] relies on matching pre-defined sensing input and retrieving pre-computed results for virtual reality (VR) applications.

However, all of these were designed for (almost) exact matching of the input (images), for specific (type of) applications only and operate within the same type of applications. In contrast, we take an unusual approach to deduplicate computation across correlated input values, across (different types of) applications, and in a fashion agnostic to the exact implementation of the processing procedures.

In this paper, we present Potluck, a cross-application approximate deduplication service to achieve the above goal. Potluck essentially stores and shares processing results between applications and leverages a set of algorithms to assess the input similarity to maximize deduplication opportunities, as detailed in Section 3. We carefully design an input matching algorithm to improve the processing performance without compromising the accuracy of the results. Potluck is implemented as a background service on Android that provides support across applications (Section 4). Extensive evaluation shows that our system can potentially reduce the processing latency for our benchmark augmented reality and vision applications by a factor of 2.5 to 10 (Section 5).

In summary, we make the following contributions:

First, we highlight deduplication opportunities across emerging vision-based and AR-based mobile applications. These arise from various sources of correlation in their input, common processing components they leverage, and the co-installation of these applications.

Second, in view of the opportunities above, we propose a set of cross-application approximate deduplication technique to achieve both fast processing and accurate results. To the best of our knowledge, this is the first such attempt.

Third, we build the system as a background service. Extensive evaluation confirms its benefit is significant.

2 Motivation

2.1 Motivating applications

Among the fastest growing applications, vision-based cognitive assistance applications and augmented reality (AR) based applications are two representative categories.

As an example cognitive application, Google Lens [10] continuously captures surrounding scenes via the camera, recognizes objects using deep learning techniques, and then presents related information to assist the user. These applications increasingly provide personal assistance.

On the other hand, AR applications such as IKEA Place [4] for home improvement, Google's Visual Positioning System for indoor navigation [3], and PokeMon Go [9] blend the virtual and physical experience. They overlay 3D graphic effects on real world scenes to enrich and enhance the interface between human eyes and the physical world.

Common themes. Most of these are lifestyle applications. According to the measurement study of the smartphone usage [34], there is a high probability of such applications being co-installed, *even though they may not be running simultaneously*. Further, they often operate in similar physical environments, share common processing steps, and map a group of similar input values to the same output. We discuss these in detail next.

2.2 Input correlation and similarity

The above applications all take input from the environment or some context, directly or indirectly. Such input exhibits similarity, within an application or across applications, due to the activities of the mobile user showing spatial and temporal correlation.

Temporal correlation. We can view the combined video input to all the applications as a continual camera feed. In other words, assuming there is a never-ending centralized camera feed to the mobile device, different applications simply take a subset of the frames as needed. From standard video analysis, significant temporal correlation exists between successive frames because the scene rarely changes completely within a short interval, and this has been leveraged extensively in video compression. In most cases, the main objects of interest in these scenes are slightly distorted versions of one another by some translation and/or scaling factor.

Spatial correlation. It is common for humans to follow along recurrent trajectories, for example, due to their regular commuting schedules or frequenting a favorite restaurant from time to time. Therefore, there is some level of recurrence of the scenes obtained as part of those activities, though potentially taken from different view points and partially different environments, such as different lighting conditions and surrounding backgrounds. The actual images might show different color bias, for example. Such correlation can be identified using SURF [12] like approaches.

Semantic correlation. A further situation arises when the same object or the same type of objects appears in completely unrelated background scenes and at different times. For example, when a road sign is detected at different places and times, regardless of the exact sign, a driver assistance app simply generates an alert. Since many applications interpret the scene to related abstract notions of objects or faces, many seemingly different images can be classified to the same category and considered semantically equivalent.

Similar but not identical. However, these correlated input frames are rarely exactly the same, for various reasons. In some cases, the scene is actually changing (e.g., the user walking or driving along a street). In other cases (e.g., approaching the same intersection from different directions), we get more or less the same scenes, but at different view angles. More generally, there might be distortion across frames due to image blur (different focus or motion-induced blur).

Correlation in the results. Generalizing the semantic correlation, these similar input values are often mapped to the

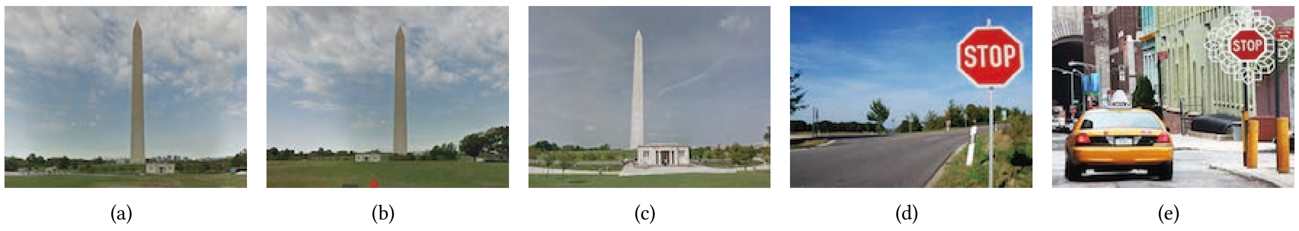


Figure 1. (a) and (b) are two snapshots taken successively along the same road 136 m apart in October 2016. (c) is taken at a similar location but in August 2014. (d) and (e) are captured in completely different places at different times, but both prominently feature a stop sign.

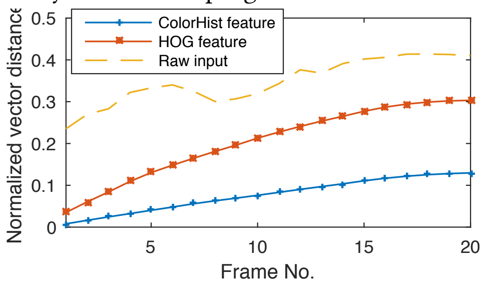


Figure 2. Similarity between frames

same output values in the aforementioned applications, due to the resolution of the results. For example, adjacent pixels in an image may be mapped to the same feature details. Image recognition functions may attach the same label to different images. For an AR application, there is no need to render a new scene if it is visually indistinguishable to our eyes from a previous one.

Examples. Figure 1 illustrates these similarities. The images are taken from Google Street View. Images (a) - (c) could be perceived "the same" by a Google Len like app for showing the Washington Monument, whereas images (d) and (e) show "stop sign".

As another example, we select a video segment from an HEVC test dataset [30], and compute several features (color histogram [22], HoG feature [45]) for consecutive frames. Figure 2 shows the relative differences between the first and later frames, calculated as the Euclidean distance between the normalized vectors of the matched features. Shorter distances indicate higher levels of similarity, although there are no universal criteria to define similarity levels. The features show consistent correlation levels across a long sequence of frames whereas the raw images do not.

2.3 Common processing steps

As noted previously [23, 33], many computer vision applications share similar, incremental processing steps. However, we also observe common processing steps in other types of applications (e.g., speech recognition) and across different types of applications (e.g., vision and AR applications).

Figure 3 shows the schematic processing flows for a cognitive application (Google Lens), and two AR based applications, *IKEA place* (as an example of AR shopping application) and *indoor navigation*.

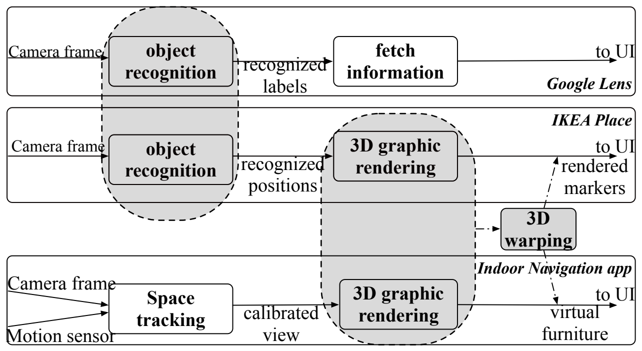


Figure 3. Schematic processing pipelines for three apps.

The indoor navigation app first recognizes the environment in the input image feed, which essentially invokes the object recognition procedure. This is also the core step of the Google Lens cognitive assistance app. Similar situations could be very common, since AR application logic typically starts with understanding the spatial context. Therefore, AR applications can share essential recognition functions with image recognition apps.

The two AR applications both require 3D graphic rendering. IKEA place would render virtual furniture at certain positions to visualize a furnished room, while the indoor navigation app would render a virtual map of merchandise to help direct customers. When the latter takes place in a furniture shop, the rendering logic would be essentially the same as what is needed for IKEA place.

Common functions are also used in non-vision based applications. For example, two location based applications can share the processing for GPS data or related contextual information close in time. A call assistant might use the mic to capture the audio to identify the location and ambient environment to determine whether to mute the call [19]. Similarly, the same procedures can be used for home occupancy detection as part of smart home management to determine whether to turn off the lights and turn on the alarm.

More generally, emerging learning-based application ecosystems further presents common APIs and libraries, and the possibility of sharing common processing steps between applications. For instance, Alexa Skills [1] enable developers to deploy various services on smart IoT devices like Amazon Echo. Deep learning frameworks such as Tensorflow [11] provide high-level programming models for app developers. Services and applications within such ecosystems leverage the same human-device interface, processing pipelines, and the underlying implementations to capture the input, understand the context, and execute tasks.

2.4 Opportunities and challenges

Given the similarities between applications discussed so far, *deduplication* is a natural approach for performance optimization. As long as these applications collectively are used frequently, there is significant potential for deduplication. For example, the home occupancy assessment and home personal assistant are usually used multiple times throughout a day; the Google Lens and indoor navigation are both likely to be used daily or more frequently.

Note that these "sharing" applications do not need to be run concurrently. Deduplication works as long as the previous results are still cached, and the interval could easily be days or longer provided there is enough space to store the cached results.

Challenges. In order to effectively deduplicate the processing across applications, we need support for identifying the equivalence between input values, cross-application sharing, and appropriate cache management criteria.

Since the input images are rarely the same, we need to be able to quantify and assess the extent of similarity between them, based on the semantics of the function.

The deduplication opportunities may straddle application boundaries, so we need a service shared between applications. This will naturally support in-app deduplication as well, though incurring a slight overhead by crossing the application boundaries.

Deduplication means we need to cache previous results. However, since our cache serves a different purpose than those of traditional caches, we cannot manage cache entries based on the least recent access or other traditional cache entry replacement algorithms.

We address these challenges by designing a cache service, Potluck, shared between applications.

3 Potluck System Design

3.1 Overview

Potluck caches previously computed results to provide approximate deduplication across applications. The processing flow is conceptually simple. When an application obtains an input (e.g., a frame from a video feed) and calls certain processing functions, it first queries the cache for any existing results. The query proceeds in several steps. First, the input data are turned into a feature vector, which serves as the key. Second, a lookup attempt is made with the key and the name of the function called, by matching the input key to any existing key within a given similarity threshold. If there is a hit, the cached result is returned. Otherwise, the application processes the raw input and then puts the result in the cache. Third, the put action may trigger an adjustment of the input similarity threshold. We discuss the individual steps next.

3.2 Computing the key

Definition of keys. The key is essentially a variable-length feature vector generated from the input image, such as SIFT [35], SURF [12], HoG [45], colorHist [22], FAST [42], and Harris [24]. For example, the feature vector might be a 768-bit vector to represent the color histogram, a vector of $N \times 64$ bytes to describe SURF features from the input image, or a vector of $m \times n$ bytes to represent the down-sampled version of the raw input image into $m \times n$ pixels.

An essential requirement is that the key must be defined in a metric space, in which a notion of distance can then be defined. This need not necessarily be the Euclidean distance, although it is the one commonly used.

Given the raw input (e.g., images or speech segments), as application requirements might differ, we give the application the freedom to choose the exact key generation and similarity assessment mechanisms. App developers can customize the implementation of both or select from a library of mechanisms provided within Potluck.

Converting the raw input to a feature vector is important, because this step can eliminate noise in the raw data, "homogenize" inputs with different formats and scales, and save space when storing these items.

3.3 The usefulness of cache entries

The *Importance* metric. Conventional caches operate within a single application, where the entries store frequently accessed data. The number of data access attempts simply reflects the value of the data and determines whether the data should be retained.

Instead, our cache is different, since not all cache entries of computation results are created equal, and the variance across applications is even larger. The access frequency is the only factor that determines the value of the cached result. Therefore, we assess the usefulness of a cache entry by a new metric, called *importance*, computed as $computation\ overhead \times access\ frequency / entry\ size$ In addition, each cache entry is tagged with a validity period. When that expires, the entry will be automatically cleared from the cache in the background.

The *importance* value indicates how frequently an entry *has been used and might save on future computation times*, but has no correlation with the accuracy of the result. Therefore, it is only used for evicting a cache entry. The lookup operation does not take into account this value.

Calculation and update of importance. The importance value for an entry is dynamic and its recalculation happens in two cases. A `lookup()` call increments the access frequency of the fetched entry by 1, and the corresponding importance value is updated accordingly. With a `put()` call, on the other hand, a new importance value is calculated for the entry. Specifically, the *computation overhead* is calculated as the elapsed time between the `lookup()` miss and the `put()` operation of this entry, and the *access frequency* is initialized to 1. The expiration time is simply that of the overall entry, set during the `put()` call.

3.4 Querying the cache

Threshold-restricted nearest neighbor query. A query involves finding the closest match for an input key. When given a feature vector as the key, we initiate a k nearest neighbour search, iterating over all entries in the key index.

After that, we discard those returned entries whose distance from the input key vector exceeds a certain threshold. By default, to balance the lookup time and quality, we set k to 1. We experimented with a few values and find that this value provides the fastest lookup time without sacrificing quality.

Random dropout. When a cache query operation is invoked, with a probability (currently set to 0.1) Potluck will simply return `null` without actually querying the cache. This is a randomization mechanism to enforce a `put()` operation at least periodically. This refreshes cache entries as well as triggers a recalibration of the threshold. The latter is valuable, in case the threshold has been loosened too much, as explained next. We will discuss how to set the "dropout" probability at the end of Section 5.2.

3.5 Tuning the similarity threshold

The threshold controls to what extent different raw inputs are consider "the same". Part of our argument is that many raw images are similar and therefore we can avoid duplicating the subsequent processing. Clearly, there is a tradeoff between performance speedup from reusing previous results and the accuracy of the results. We manage this by adaptively tuning the similarity threshold based on the ground truth and the observation of the nearest neighbour entry, as shown in Algorithm 1.

The algorithm. The idea is straightforward. The threshold is initialized to 0, meaning no distance between input images is permitted. After caching enough entries (100 by default), the algorithm kicks into action and we then gradually increase ("loosen") or decrease ("tighten") it as needed, triggered by each `put()` operation. In general, the threshold is loosened conservatively but tightened aggressively. If the threshold is too tight, we might miss deduplication opportunities. When the threshold is too loose, the cache lookups might return false positives, i.e., input images that are not actually similar but considered so due to the threshold.

Given the new key and value to be stored in the cache, the algorithm finds the nearest neighbor in the feature vector

Algorithm 1: NN-based threshold tuning algorithm

```
1  initialize threshold ← 0;
   // params are customizable
2  initialize k ← 4, α ← 0.8, z ← 100;
   Wait: z entries inserted to cache by Put operations
3  while service not terminated do
4      wait for new Put operation;
5      read (key, val) pair from the operation;
6      (key′, val′) ← lookup(key);
7      if ||key′ − key|| ≤ threshold and val′ ≠ val then
8          threshold ← threshold/k;
9      else if ||key′ − key|| > threshold and val′ = val
        then
10         threshold ← (1−α)×||key′−key||+α×threshold;
11 end
```

space to the new key. Two cases should be noted. If the key distance is larger than the threshold and both keys map to the same values (line 9 in the pseudo-code), the threshold is too tight and should be loosened with an exponentially weighted moving average. Conversely, if the key distance is within or equal to the current threshold, but the keys map to different values (line 7), the threshold is too loose and should be tightened. Note that the latter case will not arise naturally. If two keys are within the threshold, the cache query would normally return the cached result (incorrectly). Therefore we adopt the "random dropout" in the cache lookup process to artificially trigger this case from time to time, as a quality control mechanism.

Intuition and correctness. The threshold-tuning algorithm is essentially based on finding k nearest neighbors (kNN). It observes the distance between the (key, value) pairs of the nearest neighbours and compares the stored results with the ground-truth to adjust the maximum diameter of the "similar" result cluster accordingly. This diameter is then the threshold value we adopt. kNN is a widely used non-parametric, case-based machine learning algorithm, which makes no assumptions of the input data model. It has been extensively studied for decades and proven correct [27] for handling data with unknown features. In the same vein, our NN-based threshold tuning algorithm can provide reasonable hints on the correlation between input similarity and the reusability of the result even if we have no prior knowledge of the input data.

Quality of results and security considerations. While leveraging results across applications in Potluck can yield performance benefits, it breaks the isolation between applications. This can leave the system vulnerable to malicious apps polluting the cache by inserting spurious results.

Fortunately, the combination of the threshold-based kNN and random dropout algorithms can guarantee the quality of results (QoS) is not completely affected by a polluted cache and act as a defense mechanism against malicious apps. The protection can be further enhanced by incorporating a reputation system (such as Credence [47]) into Potluck. Each cache entry can be tagged with the application source. The threshold-tuning phase can then establish a reputation record for each application, and malicious apps can be identified and barred from time to time.

It is worth mentioning that sharing results in our context does not present privacy concerns. The input data tend to be derive from the contextual information for the mobile device, and hence common to all applications on the device.

3.6 Cache management

Inserting and indexing cache entries Several steps are involved to insert a cache entry (namely a `put()` operation). Potluck first collects the auxiliary information about the entry to compute its importance. Second, we invoke the threshold tuning algorithm (Section 3.5). Finally, we store the key, the computed result, and the importance value.

The key is then added to the right position in the index. The processing time depends on the index data structure. However, unlike cache lookup, the indexing process runs in the background asynchronously and does not affect the application response time.

Eviction policy and expiry. The cache entries can be discarded in two ways. First, cache entries can expire, and the timeout is currently set to be an hour. Second, if the cache is full when a new `put()` request comes, the least important entry will be evicted and replaced with the new entry.

3.7 Supporting multiple key types

So far we have explained the processing flow from a single-app (single key type) perspective. In practice, different applications may prefer to map input to feature vectors of different specifications. In other words, we need to support multiple definitions of the key, or *types*. Each application should be able to perform cache operations using their preferred key types, and we automate typecasting between keys to further support cross-application deduplication.

Multi-index structure. For multi-key-type settings, we construct a cache query index for each type of the keys, so that the query index can be optimized for the unique properties of the particular key type to ensure highly efficient lookups. Cache entries generated by different applications but using the same key type will be managed in the same index.

Cache lookup. The cache lookup will take one more argument, specifying the key type being looked up. This then sends the query to the corresponding key index.

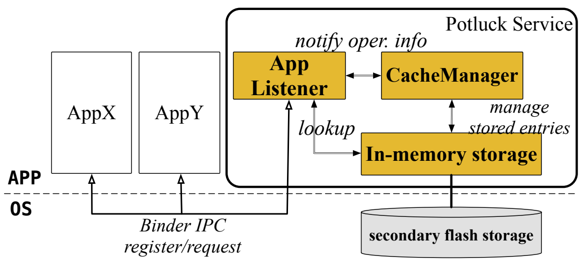


Figure 4. System architecture.

Cache insertion. Whenever a `put()` operation introduces a new key-value pair to the cache, we propagate this entry to all key indices. This triggers operations to iterate through all existing input key types, mapping the raw input to each key type, invoke the threshold tuning procedure per key index, and then insert the key to each corresponding index.

Cache entry eviction. Unlike cache insertion, cache eviction is not propagated to all indices. Instead, for each key type, the corresponding index will select the entry to be evicted and delete the key. The actual cached computation result will be cleared via garbage collection when no indices have references to it.

4 Implementation

4.1 Architecture

We implement Potluck as a background application level service in Android Marshmallow OS with API version 23. Figure 4 shows the architecture of the system.

The deduplication service consists of the following modules. The `AppListener` maintains a threadpool, handles the requests from upper-level applications, and carries out the corresponding procedures, including registering apps to the service, executing the `lookup()` or `put()` requests, and invoking the threshold tuning or reset procedure. The `CacheManager` maintains the *importance* metric of stored entries by monitoring the execution time of the functions and the access frequency of the stored entries. Based on such information, it handles the expiry and eviction in the background. The `DataStorage` is the storage layer which keeps previous computation results, and indexes the entries to speed up lookup requests.

4.2 Deduplication service

Key generation and comparison. Generally, our system supports variable-length vectors to serve as the keys. They are implemented as `Vector` instances from `java.util.Collection`, `String` instances of `java.lang.String`, or `INDArray` instances (a third-party class for fast numerical vector computation) [5]. We implement the extraction of the features mentioned in Section 3. Most of them are already implemented in the openCV library [6], and we invoke the corresponding functions to process the input image.

By default, we support comparison and similarity measurement for scalars and vectors, as well as lexical ordering and comparison for strings.

Support for custom key definition and matching. We expose an interface to the application, through which the application can customize its own key generation and comparison logic if desired. For example, app developers can implement Mel Frequency Cepstral Coefficents (MFCC) [38] computation for an audio file and Principal Component Analysis (PCA) [25] based dimensionality reduction for high-dimensional input data. Any customized classes and methods are then incorporated via *dynamic class loading*, supported via reflection in Java. In this way, app-specific components are meshed with the system logic in Potluck. Our implementation leverages the `OpenHFT.compiler` open-source package [7] to achieve this.

Cache organization. Figure 5 shows the cache layout. There are three variables, the function called, the feature vector specification (i.e., the key type), and the value of the key, that collectively correspond to a stored result. Therefore, we organize the cache entries into multiple levels, first by the functions invoked, then by the key types, and finally the specific keys.

First, when an insertion or lookup is needed, we add or match a function. This is implemented with a `HashMap`. Note that this means only applications using exactly the same function can share results. Since the type of applications that might benefit from Potluck typically use common libraries (such as OpenCV or some deep learning framework), we believe the current approach is reasonable trade-off between simplicity and effectiveness. Second, we use another `HashMap` to organize all key types corresponding to a function. Third, we use appropriate data structures to organize different key values, either a Locality Sensitive Hash (LSH) [16], KD-tree [52], treemap, or a hashmap, depending on the key dimension and how similarity assessment work. The final "values" stored are simply references (memory addresses) to the actual value stored in the memory.

A hashmap is useful for the exact matching, achieving $O(1)$ time complexity for key search. A Treemap is implemented as a balanced binary tree which supports nearest neighbor and range searches in $O(logN)$ time. Scalar or vector keys which are compared by their lexical order could benefit from using this data structure. Further, KD-trees and LSHs are data structures to support spatial indexing and efficient nearest neighbor and range searches (with $O(logN)$ average complexity) for multi-dimensional vectors, where we can only calculate distances between keys but not derive a global order for them.

Cache eviction and expiry. Cache entry eviction and expiry are handled by a separate management thread running in the background. If the cache is full when `put()` is called, the management thread will iterate through all indices to find the entry with the lowest importance to be discarded.

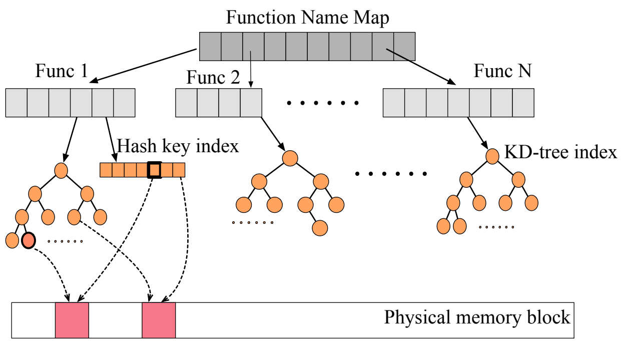


Figure 5. Cache layout.

Separately, the management thread also maintains a queue that orders all cache entries by their expiration times. This thread will be waken up when the current head item in the queue reaches its expiration time. The thread clears all (at the same time) expired entries from the cache and the priority queue, and sets the next wake-up time according to the expiration time of the new head item.

Communication between components. The communication between the apps and the deduplication service leverages Binder with AIDL [2], the IPC mechanisms natively supported by the Android OS. Interactions between the internal modules of the service are simply through shared memory with mutual exclusion locks.

The `AppListener` receives a `Request` message from an application, which consists of the request type (register or operation), function name, key type, lookup key, and computation results to store. It replies to the application with a `Reply` message containing the request type and the corresponding return values. The `AppListener` also sends the query information to the `CacheManager`.

The `CacheManager` maintains a queue of query requests. It also manages the data in the `dataStorage`, inserting new entries, evicting the least important entries when necessary, updating the *importance* value of those accessed entries, and discarding expired entries.

4.3 APIs and patches to the application code

There are two sets of APIs exposed to the application, on the control and data paths of the application respectively.

Registration on the control path. Applications start using Potluck with a `register()` call. This function registers a handle with the cache service, loads any custom-defined key generation methods, and initializes the application-specific key index. It also resets the input similarity threshold.

Cache operations on the data path. Applications can call `put()` and `lookup()`, two intuitive functions to insert and look up an entry. Therefore, the changes needed to leverage Potluckis negligible.

Discussion. Currently we need to patch the application source code to add handles to Potluck, but this makes sense because fuzzy input matching requires having the exact input values, not just their memory representations. Further, we want to expose some interface to the application to control the accuracy and performance tradeoff.

5 Evaluation

5.1 General setup

Application benchmarks. We built three simplified applications as benchmarks, one image recognition application and two augmented reality (AR) applications. The image recognition application includes pre-trained models and performs deep-learning based inference using the AlexNet neural network [29]. For the AR applications, one uses the current 3D orientation of the device and its location to render virtual objects, while the other first runs image recognition on the current frame in the camera view, and then renders virtual objects overlaid on the detected physical objects.

Data sets. While the above applications can run in real time, evaluating the recognition performance using real-time camera feeds is difficult, since it is impractical to enumerate all possible scene sequences as the input for evaluation. Further, any single camera feed only captures a single scenario, and does not necessarily represent the general case. Therefore, we turn to standard datasets used to train and test image classification algorithms. In such datasets, images are crowdsourced and well calibrated, which eliminates the spatio-temporal correlation between them. In light of this, they present less favorable (i.e., more challenge) scenarios for Potluck than datasets collected from real applications. Experiment results from these data sets are then indicative of the worst-case performance for Potluck, and we can expect better performance for real applications.

We use two commonly used image classification datasets, CIFAR-10 [28] and MNIST [31], which serves as a controlled, generic scenario. We also capture several video feeds in real life to emulate real application scenarios. The comparison between the results from these datasets cross-validates our belief that the performance of Potluck in practice will be better than reported in this section.

The CIFAR-10 dataset consists of 60,000 32×32 color images categorized into 10 classes, 6,000 each. There are 50,000 training images and 10,000 test images. We use images within the same class to mimic deduplication opportunities where similar objects appeared in different backgrounds.

The MNIST dataset is a database of handwritten digits, consisting of a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.

We found that experiment results from the two datasets were similar, and therefore we mostly present results based on CIFAR-10, as it covers a wider range of image scenes.

Experiment environment. All experiments in this section are run a Google Nexus 5 (with a quad-core 2.26 GHz Qualcomm Snapdragon as the CPU and an Adreno 330 graphics processor) as our mobile device, running Android Marshmallow OS with API version 23. Later in the section we also use a PC (with a quad-core 2.3 GHz Intel Core i7 CPU and NVIDIA GeForce GT 750M GPU) to compare the processing times. The PC is around an order of magnitude faster than the phone.

Metrics. We evaluate Potluck in terms of *accuracy*, *processing time*, and *missed opportunity*.

The first two characterize the performance benefit and tradeoff of capturing the input similarity to reduce duplicate computation. The third one is analogous to the notion of *recall* commonly used to characterize machine learning algorithms. Roughly speaking, *recall* measures the portion of test data recognized based on the training data. In our case, we first characterize the optimal case for deduplication under each specific experiment setting, which defines the upperbound performance of our system, and then quantify *missed opportunity* by the gap between the performance of Potluck and the particular optimal case.

Since the input is the main determining factor for the performance of our system, our results are interpreted with respect to the input data setting of each experiment.

5.2 Input and key management

Key generation. We randomly select a set of 600×400 images from our dataset, and measure the time taken to generate a feature vector following different feature extraction methods. Around 500 features are detected in each image. Table 1 shows that generating SIFT and SURF features as the key takes orders of magnitude longer than the others but captures more information about the raw image. They are suited to recognition tasks. Harris and FAST features are based on edge detection and a good fit for object detection workloads. Detection is the first step of recognition, and the latter requires much more detailed information. *Downsamp* refers to down-sampling the raw image to fewer dimensions, which is then vectorized to be fed into deep neural networks. Since key generation is the first step to use Potluck, there is clearly a tradeoff between the processing time and the level of feature expressiveness required for a specific app. For our later experiments, we use *Downsamp* for the deep learning based image recognition app and FAST for motion estimation within the AR applications.

Threshold tuning. The similarity threshold determines the amount of deduplication we can achieve as well as the accuracy of the lookup result. The threshold is loosened or

Table 1. Key generation time

Feature	Size (KB)	Time (ms)	Usage
SIFT	124	1568	Recognition
SURF	32	446	Recognition
Harris	35	91	Detection
FAST	28	4.6	Detection
Downsamp	1	5.8	Deep learning

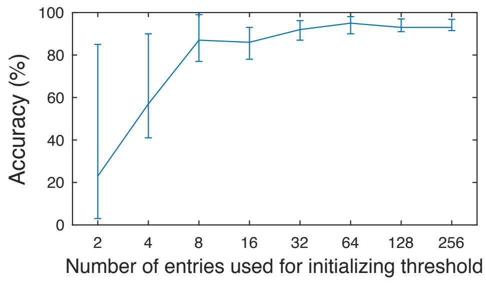


Figure 6. The accuracy of the similarity threshold.

tightened depending on the cached entries. We perform two experiments to evaluate the tuning algorithm.

First, we investigate how many entries should be cached before we start calibrating the threshold and thus *enabling* the deduplication service. We consider a threshold "accurate" if the results from the cache query are similar to the ground truth. We randomly pick a variable number of images from the training set of CIFAR-10, put the recognition results into the cache, and calculate the initial value of the threshold. Then, we take 400 images from the test set and obtain the recognition results by both running the recognition algorithm and retrieving the nearest match from the cache results. These steps are repeated 10 times and we collect the average and variance information.

Figure 6 shows the *normalized recognition accuracy* of the threshold vs the number of cache entries used for initializing the threshold. Since the recognition accuracy without leveraging deduplication is not 100% anyway, we use that as a baseline to normalize the accuracy of our system. In other words, the y-axis shows the accuracy with Potluck divided by the baseline accuracy value. The line shows the average value, while the errorbars show the maxima and minima.

The accuracy stabilizes quickly as more cache entries are available. With at least 32 entries (over 1 second for a normal 30 fps video feed), the accuracy exceeds 95% with less than 5% error. The time overhead for computing a new threshold turns out to be less than 1 ms and negligible.

Second, we analyze how quickly the threshold is tightened. Recall that we loosen the threshold slowly and conservatively to minimize the possibility of false positives, but try to tighten it quickly. This is also because the threshold is loosened more frequently, invoked by each natural `put()` operation. In contrast, it is only tightened after a random dropout mechanism (Section 3.4), which happens rarely. In this experiment, we start with a certain threshold (normalized to 1), and then count how many cache entries are needed to adjust the threshold to 0.

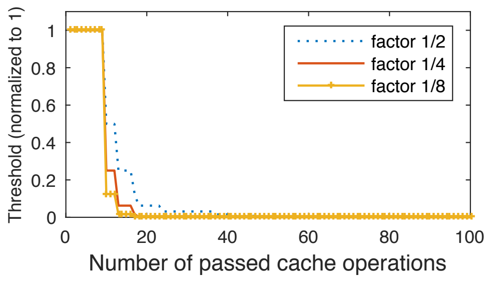


Figure 7. Threshold changes with lookup operations.

Figure 7 illustrates that, when the decrease factor (the parameter k defined in Alg. 1) is over 1/4 and the dropout probability 0.1 (the respective default value used), within around 20 cache operations (including `lookup()` and `put()`), the threshold shrinks by a factor of 20. With only 30 operations on average, we could further shrink it by a factor of 100. In other words, when switching to a new scene, for a 30 fps camera, the threshold could be adjusted accordingly within seconds, which is an acceptable latency for most use cases.

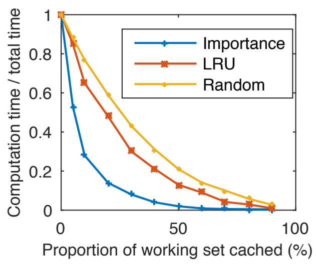
5.3 Cache entry replacement strategy

To evaluate our cache replacement strategy, we consider two cache hit patterns, uniform distribution and exponential distribution, and compare our importance-based strategy with two commonly used cache replacement strategies, *least recently used* (LRU) and *random discard.*

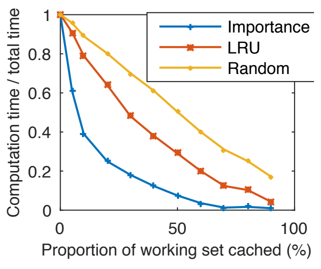
The number of cache hits, or the occurrences of reusable results can be modeled by a uniform distribution or an exponential distribution. Uniform distribution is often seen for single-app or in-app deduplication, as it is common to obtain input frames at fixed intervals and each component in the processing pipeline is invoked once per new input. Exponential distribution fits the multi-application scenario as the relative application popularity can be modeled by an exponential distribution [17].

For this experiment, we first define 100 different workloads, each of which takes a different amount of computation time ranging from 1 ms to 10 s. Then we create two request arrival sequences with 10,000 requests each, generated from these 100 workloads. Within the two request sequences, the number of occurrence of each workload is uniformly and exponentially distributed respectively. Next, we vary the proportion of working sets cached from 10 to 90 (meaning caching 10% - 90% of all workloads). Under each value, we submit the request sequence to the cache and measure the portion of the total computation time required due to cache misses, using the three different cache replacement algorithms.

Figure 8 shows that our algorithm consistently outperforms LRU by a large margin. For both request patterns, using the importance metric to retain entry caches can save an additional 40% of the computation while caching less than 20% of the previous results. The fraction of computation time drops further to below 5%, when the proportion



(a) Exponential distribution



(b) Uniform distribution

Figure 8. Comparison of cache entry replacement strategies given different access patterns.

Table 2. Lookup latency

# of entry	key size (bytes)	LSH (μs)	enum (μs)
100	100	3.2	50
1000	100	3.6	170
10000	100	4.4	2210
100000	100	6.7	21340
100000	1000	7.5	205070
100000	5000	8.1	–

cached grows to over 40% and 60% of the active working set respectively for the exponential and uniform workload distributions. The non-uniform distribution in the request patterm will propagate to the importance values, skewing the distribution of the latter. These results suggest that our algorithm successfully retains the results from the computation-intensive workloads.

5.4 System overhead

Cache lookup and insertion overhead. The cache lookup overhead depends on the organization, the current cache size, and the key length. The computation complexity of a key matching operation is determined by the key length. We submit 100 requests to the cache and measure the average completion time.

Table 2 compares the lookup time using Locality Sensitive Hash (LSH) and by naively enumerating through all keys. LSH based lookup is very efficient, takes less than 10μs, and scales well with an increasing cache size. Without a carefully designed key structure, the best one could do is to resort to naive enumeration. It incurs an acceptable latency when the total size of the keys is no larger than 10 MB, but cannot scale well to hundreds of MB.

The insertion overhead is at micro-second level even for a 500 MB cache (about the upper limit, since using more space is not practical for mobile devices), which is negligible.

IPC latency. We sequentially submit 500 requests and divide the total response time by 500. The average end-to-end latency using the Binder and AIDL mechanism is about 0.36 ms per request.

Space overhead. Android sets a per-device limit for the maximum heap space an app could use, ranging from 16 MB to 512 MB. This simply prevents applications from exhausting the memory, and our service operates within the limit.

Our key structure is also efficient regarding space usage. Consider a raw image of 400×400 pixels, about 500 KB in size. Its SIFT or SURF feature vectors are only 48 KB and 24 KB in size when 400 features points are extracted. Other feature vectors (such as FAST features) are often more compact. Even if all these vectors are used simultaneously, their combined size is still an order of magnitude smaller than that of the raw image.

Further, as we mentioned previously, even though we use multiple key indices, the corresponding "values" are only memory addresses, not the actual recognition results. This way, the recognition results are not stored redundantly.

Generalization to other hardware models. The overhead numbers listed above further imply that the benefit of Potluck is not device or CPU dependent, but bound by the input. The read/write speed for memory and flash storage access varies little across phone models, while the latency due to the lookup/pipeline overhead is at least 3 orders of magnitude lower than running the same computation on a high-end GPU-equipped device. In fact, we will show later (Section 5.6) that an old phone running deduplication could outperform a powerful PC.

5.5 Single-application performance

We next evaluate the end-to-end performance of Potluck for applications individually. We run both the deep learning application and the AR application that loads and renders 3D models based on the current location and device orientation.

Performance and accuracy tradeoff. We randomly select 100, 500, and 5000 images along with their (ground-truth) recognition labels from the CIFAR-10 training set and 500 images from the MNIST dataset as the pre-stored entries, and then select 100 images from the test set as the inputs to the cache lookups. Figure 9 shows the processing time saved and the accuracy respectively as the threshold changes. The actual threshold values produced by our tuning algorithm stay within the shaded region in either figure.

The performance of Potluck is measured by dividing the accuracy and time saving by the respective optimal values. The optimal accuracy is defined as the accuracy when using the pre-trained AlexNet deep neural network to recognize the test images. The optimal time saving is 100% assuming all lookups result in cache hits (with the right results).

We make several observations from the figures. First, our threshold tuning algorithm results in a reasonable tradeoff by saving up to 80% of total the computation time at the expense of less than 10% accuracy drop. Second, when there is a larger number of stored results, the accuracy starts to drop slightly earlier. This makes sense because more cached results can increase the noise and the chance of mis-classification (i.e., false positives for key matches). Third, not surprisingly, the

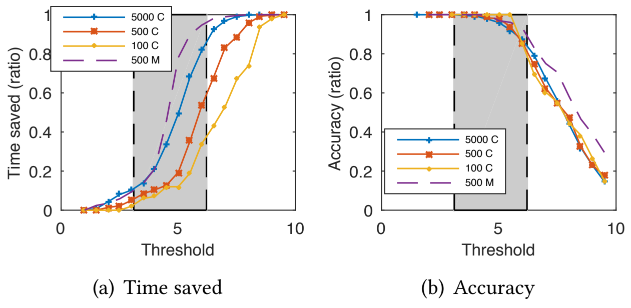


Figure 9. Time saving and accuracy vs the extent of deduplication opportunities. "C" and "M" correspond to the CIFAR-10 and MNIST datasets respectively

total time saving increases significantly with the number of stored entries, as there is a higher probability of cache hits. Lastly, the two different datasets shows consistent trends for the tradeoff between the accuracy and total time saved. This suggests the generic behavior of the system under various scenarios.

Note that we cannot assess the "accuracy" easily for AR applications, since the rendered scenes are evaluated with a number of visual quality metrics, such as the image resolution, and there is no absolute notion of "accurate".

Mobile processing vs offloading to a PC. To further gauge the benefit of Potluck, we compare the application processing time on the mobile device and on the PC mentioned earlier. The latter is a proxy for offloading computation to a power server but *without incurring network transfer latency.*

For the *deep learning based image recognition application*, the experiment setting is the same as for the previous experiment, except that we run our threshold tuning algorithm live to automatically adjust the threshold, instead of fixing its value. Figure 10(a) shows the normalized average completion time for each image with optimal deduplication, with and without Potluck on the mobile device, and on the PC. The performance of Potluck is within 5 ms of the optimal case. It reduces the completion time of the native application on the mobile by a factor of 24.8, and even reduces the native execution time on the powerful laptop PC by a factor of 4.2.

For the *3D graphic rendering* part of the AR application, our target results are three 2D scenes with depth information, each containing virtual 3D objects of different rendering complexity. Normally, a 3D object is rendered and then projected onto the display. With Potluck, the processing flow is simplified to looking up rendered 2D images with the most similar orientation, estimating the transform matrix, and warping the original 2D image to fit the current orientation [36]. The 3D orientation and location of the device are used as the key for the cache lookups in Potluck.

Since we only care about the transform matrices between scenes, and not the actual scenes in the video, for this experiment we generate a video feed of three virtual 3D models viewed from different angles and sample non-consecutive frames to synthesize the workload to emulate a real scenario.

Figure 10(b) compares the per-frame rendering time needed by Potluck with the times for native rendering on the mobile, on the PC, and the optimal deduplication case. The performance of Potluck is within 9.2% of the optimal deduplication performance. It reduces the running time of the native application on the mobile device by a factor of 7, and only takes 47% longer than rendering on the PC.

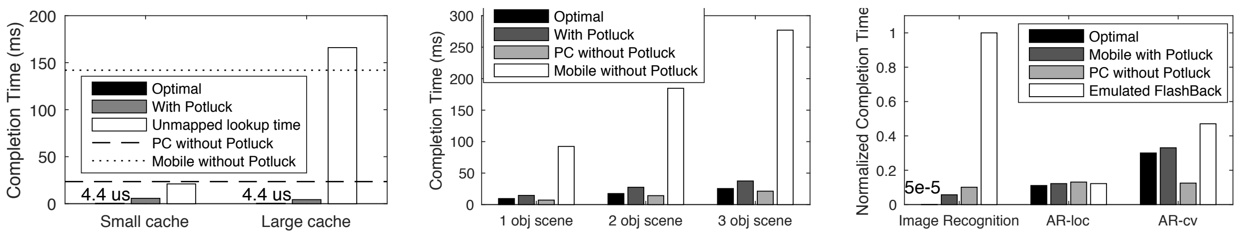
5.6 Multi-application performance

Finally, we run the three applications together, two AR applications and an image recognition app as described at the beginning of the section. Note that the applications are not required to run simultaneously in the foreground in the classical sense of concurrency. Rather, we emulate a scenario where the invocations of these applications are interleaved in similar spatio-temporal contexts. We record several 30-second video segments from the real world at 60 fps, extract 200 frames, evenly spaced, from each video sequence, as our input sequences, and evaluate the performance gain from Potluck.

Figure 10(c) shows the normalized completion time of the three applications respectively. Potluck reduces the per-frame completion time by 2.5 to 10 times, and almost achieves the same performance as optimally reusing previous results. For the deep learning and the location based AR applications, running deduplication on the mobile device is even faster than running the whole workload on the PC.

The last set of bars in the figure represents an emulated version of FlashBack [14]. This is a system to achieve fast graphics rendering for *virtual reality* applications, by precomputing all possible input combinations and simply looking up the corresponding results during the actual run. This is the closest to reusing previous results for AR applications, even though the input handling is different. Assuming the same result from the input handling techniques in FlashBack and Potluck, the benefit of FlashBack only extends to in-app result reuse for only the rendering portion of our AR applications. In view of our benchmark applications, therefore, the emulated FlashBack can benefit the location-base AR application similar to Potluck does, benefit the rendering portion of the second AR application, but does nothing for the deep learning application.

We also evaluated Potluck on the MNIST dataset. The images in this set show higher semantic correlation than those in CIFAR-10. While Potluck delivered a similar time saving for the location-based AR workload as shown in Figure 10(c), it reduced the processing time of the image recognition application by a factor of 16 compared to native computation on the phone. This highlights the benefit of Potluck when the input data exhibit stronger correlation, as Potluck is able



(a) Deep learning (b) AR rendering (c) Three apps concurrently

Figure 10. Time saving and accuracy vs the extent of deduplication opportunities.

to eliminate more potentially duplicated processing. Further, these results again suggest that Potluck could bring significant performance gain in a broad range of scenarios.

6 Related Work

To the best of our knowledge, no existing work has explored cross-application approximate deduplication in mobile scenarios. Further, Potluck provides a more generic mechanism for deduplication even within the same (type) of applications. We discuss a few approaches closest to ours.

Application-specific solutions. Starfish [33], Flashback [14], and MCDNN represent recent efforts accelerating computation-intensive mobile applications. **Starfish** extends common computer vision (CV) libraries [6] with a centralized mechanism, including a cache to store previous function call arguments and results. However, it does not readily work for newer DNN-based applications [13, 49], and its memoization requires precise matching between the inputs. **FlashBack** is a pre-rendering system specifically designed for virtual reality (VR) applications. It utilizes nearest neighbour matching to select pre-rendered frames and adjust them for new frames. However, the design assumes fixed environment and known data pattern, which is not the case for non-VR applications, as the scenes for AR applications are unbounded and constantly changing. **MCDNN** accelerates the execution of deep neural networks on mobile devices. One particular optimization is to share the execution (results) of the common layers of the neural networks from different applications. But, the sharing is synchronous and does not involve explicit caching. If the exact same input is passed to the neural network twice, the whole computation will be performed twice.

In contrast, Potluck is more flexible in several ways. It targets cross-application deduplication, does not require applications to run concurrently to share results, and makes little assumptions of the specifics of the sharing applications or the shared input data. The input similarity is determined semantically, rather than based on the raw binary representation.

Deduplication vs frame sampling. Though not designed for the same settings, several previous efforts related to video analytics [51] or continuous vision [15] considered some form of *frame sampling* to reduce the computation complexity. These systems selectively process "the most interesting" input frames and skip the rest. They include algorithms to identify the frames of interest for further processing.

Potluck can be viewed as a different take on frame sampling. Our service computes the full results for selected input images and find a nearest match for the rest. One important difference is that Potluck makes no assumptions about the sequence of input images and is more flexible for applications launched in an ad hoc fashion.

Computation reuse in clusters. In distributed clusters, Differential dataflow reuses results between iterations within the same program [37], while DryadInc [41], SEeSAW [26] and Nectar [20] leverage cross-job (not application) computation reuse with a centralized cache service and a program rewriter that replaces redundant computation with the corresponding cached results. These solutions do not readily apply to a mobile setting, since the resource constraints are completely different, and there is no distributed filesystem as a basic structure to synchronize data globally. UNIC [46] is a recent work specifically designed for deduplication security, which is orthogonal to our design. The techniques proposed by UNIC can be incorporated into Potluck for better program-level integrity and secrecy.

Approximate caching. There is a loose analogue between deduplicating *computation* in Potluck and deduplicating *storage* in approximate caching (such as Doppelg´anger [39]) and the compression of image sets [50] or nearly identical videos [48]. All cases are motivated by the similarity between input images, and various feature extraction mechanisms can be used to quantify the similarity for further compression. However, Potluck further reasons about the computation resulted from the input similarity, whereas the other schemes aim to reduce the space usage of the input.

7 Conclusion

In this paper, we argue for an unorthodox approach to optimize the performance of computation-intensive mobile applications via cross-application approximate deduplication. This is based on the observation that many emerging applications, such as computer vision and augmented reality applications take similar scenes as the input and sometimes invoke the same processing functions. Therefore, there are ample opportunities for reusing previously computed results.

We build Potluck, a background service that conceptually acts as a middleware to support multiple applications. Potluck converts the input image to a feature vector, which, along with the function invoked, then serves as a key to the previously computed result. The design further includes mechanisms to dynamically tune the input similarity threshold and manage cache entries based on their potential for reuse. Evaluation shows that we can speed up the processing significantly via deduplication.

Looking ahead, we believe there is scope to explore further deduplication opportunties. In this paper we have mainly focused on image-based applications, since they tend to be among the most computationally intensive. However, the design and implementation presented are general and can apply to other types of input data. We can also apply the deduplication concept across devices. Further, the applications could exploit optimization opportunities by adding post-lookup logic to perform incremental computation.

References

[1] Alexa Skills Kit: Add Voice to Your Big Idea and Reach More Customers. https://developer.amazon.com/alexa-skills-kit.

[2] Android Interface Definition Language for IPC. https://developer.android.com/guide/components/aidl.html.

[3] Google's indoor VPS navigation by Tango-ready phone. http://mashable.com/2017/05/google-visual-positioning-service-tango-augmented-reality/.

[4] How stores will use augmented reality. https://www.technologyreview.com/s/601664/.

[5] N-Dimensional Arrays for Java. http://nd4j.org/.

[6] Open-source computer vision. http://opencv.org/.

[7] Openhft java runtime compiler. https://github.com/OpenHFT/Java-Runtime-Compiler.

[8] Openshade with visually impaired users. http://www.openshades.com/.

[9] PokeMon Go augmented reality game. http://www.pokemongo.com/.

[10] World around you with Google Lens and the Assistant. https://www.blog.google/products/assistant/world-around-you-google-lens-and-assistant/.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[12] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[13] G. Bertasius, J. Shi, and L. Torresani. Deepedge: A multi-scale bifurcated deep network for top-down contour detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4380–4389, 2015.

[14] 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.

[15] 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.

[16] 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.

[17] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[18] J. Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012.

[19] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the Annual International Conference on Mobile Computing and Networking, MobiCom16*. ACM, 2016.

[20] 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.

[21] 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*, MobiSys '14, pages 68–81, New York, NY, USA, 2014. ACM.

[22] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE transactions on pattern analysis and machine intelligence*, 17(7):729–736, 1995.

[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*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[24] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[25] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[26] K. Kannan, S. Bhattacharya, K. Raj, M. Murugan, and D. Voigt. Seesaw-similarity exploiting storage for accelerating analytics workflows. In *HotStorage*, 2016.

[27] J. M. Keller, M. R. Gray, and J. A. Givens. A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics*, (4):580–585, 1985.

[28] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[29] 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.

[30] J. Le Feuvre, J. Thiesse, M. Parmentier, M. Raulet, and C. Daguet. Ultra high definition HEVC DASH data set. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 7–12. ACM, 2014.

[31] Y. LeCun, C. Cortes, and C. J. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[32] 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*, MobiSys '15, pages 151–165, New York, NY, USA, 2015. ACM.

[33] 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.

[34] X. Liu, H. Li, X. Lu, T. Xie, Q. Mei, H. Mei, and F. Feng. Understanding Diverse Smarpthone Usage Patterns from Large-Scale Appstore-Service Profiles. *arXiv preprint arXiv:1702.05060*, 2017.

[35] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[36] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46. ACM, 1995.

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

[38] P. Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116:374–388, 1976.

[39] 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.

[40] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 276–287, New York, NY, USA, 1997. ACM.

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

[42] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

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

[44] A. Shashua, Y. Gdalyahu, and G. Hayun. Pedestrian detection for driving assistance systems: Single-frame classification and system level performance. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 1–6. IEEE, 2004.

[45] F. Suard, A. Rakotomamonjy, A. Bensrhair, and A. Broggi. Pedestrian detection using infrared images and histograms of oriented gradients. In *Intelligent Vehicles Symposium, 2006 IEEE*, pages 206–212. IEEE, 2006.

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

[47] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *USENIX NSDI*, volume 6, 2006.

[48] H. Wang, T. Tian, M. Ma, and J. Wu. Joint Compression of Near-Duplicate Videos. *IEEE Transactions on Multimedia*, 2016.

[49] Z. Wang, D. Liu, J. Yang, W. Han, and T. Huang. Deep networks for image super-resolution with sparse prior. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 370–378, 2015.

[50] H. Wu, X. Sun, J. Yang, W. Zeng, and F. Wu. Lossless Compression of JPEG Coded Photo Collections. *IEEE Transactions on Image Processing*, 25(6):2684–2696, 2016.

[51] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438. ACM, 2015.

[52] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.