Adaptive Distributed Caching for Scalable Machine Learning Services

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Utsav Drolia

B.Eng., Electronics and Communication, Manipal University
M.S., Embedded Systems, University of Pennsylvania

Carnegie Mellon University
Pittsburgh, PA

August 2017

Acknowledgements

First and foremost, I would like to thank my parents for making it possible for me to travel to a different country and encouraging me to study further. None of this would be possible without their blessing and support.

I would like to thank my friend, ally and wife, Megha for being the pillar, the shoulder and the cheerleader throughout this journey. Thank you for believing in me when I didn't, and for being there whenever I needed a sympathetic ear.

I would like to thank my advisor and my guide, Prof. Priya Narasimhan, for allowing me into her research group, and into CMU. Thank you for this opportunity and for believing in me. Thank you for teaching me how to conduct research, how to think about problems, put my ideas into words and to present them in a coherent manner. Your passion for research and sheer capacity for hard work have been an inspiration.

I would like to thank members of my thesis committee, Prof. Dan Siewiorek, Dr. Rajeev Gandhi, and Dr. Marina Thottan for guiding my research and providing insightful feedback. Thank you for making time to go over my dissertation and for my defense. I would also like to thank Dr. Ravi Iyer for his feedback and support.

I would like to thank my mentor during my internship at Bell Labs, Dr. Katherine Guo. Thank you for letting me chose my internship project and for guiding me during, and even after my internship.

I would like to thank Dr. Rajeev Gandhi for providing technical feedback on every project I undertook over my years at CMU. I would also like to thank all of my colleagues at CMU - Jiaqi, Nathan, Rolando, Kunal and Soila. Soila, thank you for guiding me through my first substantial Ph.D project, and my first significant publication. Kunal, thank you for your enthusiasm and encouragement. Rolando, thank you for introducing me to different fields of distributed systems research. Jiaqi and Nathan, thank you for being a sounding board for my ideas.

I would like to thank faculty at CMU - Phil Koopman for teaching me about software for embedded systems, Tom Mitchell for teaching me about machine learning, Bill Nace for introducing me to distributed systems, and Dan Siewiorek and Asim Smailagic for teaching me about building end-to-end systems in teams.

I would also like to thank Karen Lindenfelser, Joan Digney, Jennifer Zeni, Toni Fox, Nathan

Snizaski, and Samantha Goldstein, who have allowed me to focus on my research while depending on them for any administrative support.

I would like to thank my friends in Pittsburgh for being a second family.

During my dissertation research, I received financial support from the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and Embedded Computing (ISTC-EC), Carnegie Mellon's CyLab and Parallel Data Lab, and from the CMU-Portugal Program. I also received support in the form of equipment and data from YinzCam Inc.

Abstract

Applications for Internet-enabled devices use machine learning to process captured data to make intelligent decisions or provide information to users. Typically, the computation to process the data is executed in cloud-based backends. The devices are used for sensing data, offloading it to the cloud, receiving responses and acting upon them. However, this approach leads to high end-to-end latency due to communication over the Internet.

This dissertation proposes reducing this response time by minimizing offloading, and pushing computation close to the source of the data, i.e. to edge servers and devices themselves. To adapt to the resource constrained environment at the edge, it presents an approach that leverages spatiotemporal locality to push subparts of the model to the edge. This approach is embodied in a distributed caching framework, Cachier.

Cachier is built upon a novel caching model for recognition, and is distributed across edge servers and devices. The analytical caching model for recognition provides a formulation for expected latency for recognition requests in Cachier. The formulation incorporates the effects of compute time and accuracy. It also incorporates network conditions, thus providing a method to compute expected response times under various conditions. This is utilized as a cost function by Cachier, at edge servers and devices. By analyzing requests at the edge server, Cachier caches relevant parts of the trained model at edge servers, which is used to respond to requests, minimizing the number of requests that go to the cloud. Then, Cachier uses context-aware prediction to prefetch parts of the trained model onto devices. The requests can then be processed on the devices, thus minimizing the number of offloaded requests. Finally, Cachier enables cooperation between nearby devices to allow exchanging prefetched data, reducing the dependence on remote servers even further.

The efficacy of Cachier is evaluated by using it with an art recognition application. The application is driven using real world traces gathered at museums. By conducting a large-scale study with different control variables, we show that Cachier can lower latency, increase scalability and decrease infrastructure resource usage, while maintaining high accuracy.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Devices and sensors are everywhere, and more are coming online every day. They range from small sensor-based devices that are part of the Internet-of-Things, to the ubiquitous mobile devices like smartphones and tablets, to wearables, to flying drones and even cars. All of these devices are embedded with sensors which generated volumes of data. The goal of these devices is to provide users or enterprises with concise, actionable information. Hence, the raw data generated by the sensors needs to processed and analyzed to glean actionable intelligence. This is where machine learning and recognition algorithms have been immensely useful. They have enabled devices to be more autonomous and/or provide users with a better understanding of the world around them. Among these algorithms, image recognition has enabled a number of new applications. They have been applied in cars to provide situational awareness and autonomous functionality [2]. They form the basis for many applications on users' mobile and wearable devices. They enable indoor navigation [3], smarter shopping [4], painting recognition [5], augmented museums [6], social network applications [7, 8], and much more. They have lead to a new class of wearable devices as well - head mounted displays for augmented reality (AR) such as Google Glass [9] and HoloLens [10]. Machine learning and image recognition is transforming the way devices and humans perceive the world.

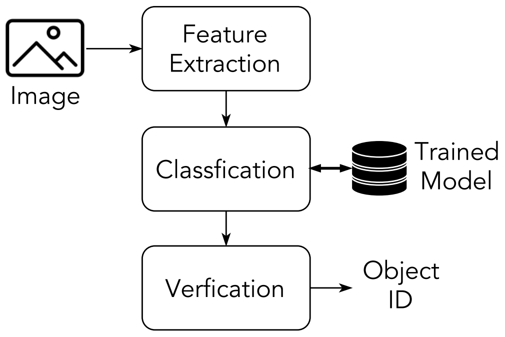
Today, image recognition applications rely on machine learning models that are trained in the cloud, while inference requests are made at run-time using edge devices. These applications require large sets of data to train the models required for image recognition tasks and formidable compute resources to provide accurate and timely results. Since such training data is stored in the cloud, and the cloud provides vast amounts of compute resources, training of these models are typically carried out in the cloud. Given the availability of these compute resources alongside trained models in the cloud, inference on image recognition requests is fast as well. Thus, recognition applications, especially those that target mobile, wearable and other embedded devices[1], which are typically considered resource constrained, rely on an offloading model. They offload inference tasks to the cloud. This has also prompted the rise of cloud-based services that offer recognition as a service [11, 12, 13, 14, 15, 16]. Here, the edge device captures images/video-frames and sends them to the service over the Internet, which performs inference on the image/video-frame and sends back relevant information based on the result of the inference.

For applications that rely on recognition, a short response time is important. For example, if the application is taking an action based on the response, e.g. in drones or autonomous vehicles that rely on recognition to navigate, then responses will have extremely short deadlines [17]. Or if the application is providing information or content to users, e.g. mobile augmented reality applications or product information applications, then responses need to be quick so that the information can be correctly overlaid on the camera's live view. In the current offloading model, the response time for a request from the edge device includes not just the compute time in the cloud, but also the time it takes to send the images over the network and receive the response. Even if the inference algorithm is fast and the cloud provides vast resources for inference, the latency incurred due to the network can still cause high response times. Moreover, the projected increase in the number of devices that can use such applications is likely to aggravate this problem further [18]. Millions of devices, each sending streams of images, across the Internet, to a centralized cloud-based service, will likely cause the Internet bandwidth to become a bottleneck [19]. Thus, simply offloading requests to the cloud is not a panacea and certainly cannot enable large scale, real-time recognition.

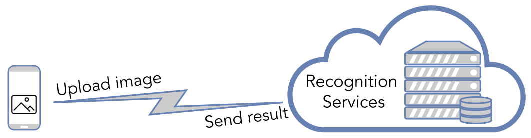
1.1 Background

Image recognition or recognition refers to the mechanism of identifying objects in images or frames of a video sequence. As shown in Figure 1.1a, image recognition works by connecting different algorithms and techniques in a pipeline. Here we highlight the main components of

[1]We will refer to these devices as edge devices.



(a) Image recognition pipeline.



(b) Mobile image recognition through offloading.

Figure 1.1: For mobile image recognition, edge devices send their image requests to recognition pipelines, shown in 1.1a, running in the cloud (1.1b).

this pipeline that are used at run time, i.e. when an inference has to be made. This pipeline of extraction, classification and verification is typical for image recognition [20].

1.1.1 Feature Extraction

Features are numerical descriptors that individually or collectively form unique signatures that describe the image. Typically, image recognition techniques rely on discovering multiple interesting points in the image and extracting descriptors for these points. These interesting points collectively form a signature for the image. This technique is used by many feature detection and extraction techniques such as SIFT [21], SURF [22] and ORB [23]. Recently, convolutional neural networks (CNN) have also been used for feature extraction [24]. CNNs describe entire images, not points in images.

1.1.2 Classification

Extracted features are then classified into classes representing the objects that need to be recognized. A trained model is used for this task. A model is trained offline using features extracted from training data. The more classes, or objects, that need to be recognized, the larger this model typically is. Different types of models have been suggested in the literature, for example Locality Sensitive Hashing [25], kd-trees [26], Support Vector Machines [27] and deep neural network (DNN) approaches [24].

1.1.3 Verification (optional)

After the feature classification, some techniques employ verification, typically through geometric algorithms. These are especially useful for AR-type applications since this verification provides the location of the object being recognized within the input image.

Applications on edge devices that use these techniques, typically do not run this pipeline locally on the devices. Instead, they offload the execution to remote services that run the pipeline, as discussed earlier, and shown in Figure 1.1b. The device uploads the images being captured to the remote service and receives the recognition results. These remote services can be in the cloud, across the Internet, or in the edge servers, closer to the user. Typically applications do not present the recognition result (the name or ID of the object recognized) itself to the user. Instead, they map the recognition result to some form of content or information that is presented to the user.

1.2 Towards large-scale real-time recognition for edge devices

There have been multiple different approaches that address challenges related to real-time recognition on edge devices. These approaches can be broadly divided into two large categories: (a) algorithmic approaches, and (b) systems approaches. Approaches related to algorithmic innovation propose new recognition algorithms such as faster feature extraction or new models for classification. Whereas systems approaches take existing algorithms and propose how best utilize and deploy them to ensure low latency. This thesis falls under the second category.

The approaches that propose new systems can be further divided into different categories for closer examination. First, some research efforts propose enabling mobile-cloud offloading techniques that allow devices to offload computation to the cloud. These approaches use different applications as motivation for their work, of which image recognition is one. Second, some research efforts propose a new computing paradigm that extends cloud computing and brings cloud-like services closer to the edge. These approaches also use image recognition as a motivating application and show how such applications can gain lower response times when offloading to these closer cloud-like compute resources. Third, another category of research efforts takes a more specific approach. These efforts propose to optimize the offloading model proposed by previous approaches to lower response times specifically for image recognition applications. They do so by leveraging properties of image recognition applications. Note, the common theme across all of these systems approaches are that they still rely on offloading, i.e. the inference task is only run on one remote location. Moreover, they do not consider the impact of scale on such applications and systems. On the other hand, the caching approach proposed in this thesis takes an adaptive approach. It proposes to partition and move the inference task fluidly across devices, edge servers and the cloud, and thus use all the resources available to lower latency. The partitioning and placement can adapt to run time conditions. This approach is designed to address the impact of scale and leverage the opportunity provided by it as well.

1.2.1 Algorithmic approach

There have been advances in the field of computer vision and image recognition algorithms to enable efficient processing locally on mobile and edge devices. Feature extractors and descriptors such as SURF [22] and ORB [23] have certainly made it possible to carry out image recognition on devices. However, as reported in [28, 29], local recognition does not scale beyond tens of images, which is not enough for practical applications, where the classes of objects can number in the thousands. Deep learning and neural networks have made it possible to achieve high recognition accuracy [30], and it has been achieved on mobile devices as well [31], but again the device alone cannot achieve the diverse recognition at the required speed [32].

Given that mobile devices alone cannot support recognition of more than a few categories, they still need to augment their local resources with a backend server, typically in the cloud.

1.2.2 Mobile-cloud offloading approach

There are several systems that have been proposed to offload computing from mobile devices onto backend servers, in the cloud, to enhance applications. These systems divide the application at different granularities to offload parts of it to the cloud, optimizing for different objectives.

MAUI [33] optimizes for energy consumption. It profiles applications to know the energy costs of running functions of the application on the device and of sending data over wireless networks to the cloud. Based on this offline and online profiling, it offloads specific functions, indicated as "offloadable" by the developer, from the application to the cloud if it predicts that it would lower energy costs. Interestingly, as a by-product of the offloading, their evaluation shows that latency is lowered as well.

CloneCloud [34], on the other hand, optimizes for latency. It measures and profiles run times of threads of execution in the application instead of functions. This allows it to offload threads to the cloud without developer hints.

Both approaches use a simple image recognition application, face detection, as an example application for evaluation and show that their respective systems lower energy and response times when a single device offloads to a remote server. This application does not require a complex model since it is detecting faces, not recognizing them. However, the resulting response times are still not low enough to enable real-time recognition and truly interactive applications. Moreover, the evaluations do not explore the effect on the network and the remote resources if multiple devices try to offload simultaneously.

1.2.3 Edge computing approach

The issue of mobile-cloud computing not being able to keep up with latency-sensitive recognition applications has been highlighted in the literature [35, 36, 37, 38]. The key issue is network latency. Since remote servers in the cloud need to be accessed by traversing the Internet, mobile-cloud computing incurs high latency due to the network. Thus, this new body of work proposes that for such latency-sensitive applications, it will be better if the cloud is brought "closer" to the devices.

Satyanarayanan et al. [36] propose the idea of a "cloudlet", a server co-located with last-mile networking infrastructure such as cellular basestations or Wi-Fi access points and controllers. Such a server can potentially function as a mini cloud, sitting in-between the device and the cloud. The interaction with these cloudlets is designed to in the same manner as in mobile-cloud systems. A surrogate VM can be deployed on a cloudlet by a device, which can run application software for the device. The device is then only used for input and output, while all the heavy lifting is done by the cloudlet. Ha et al. [38] goes on to show how such edge servers can help decrease latency for specific multimedia applications, compared to using the cloud. Their evaluation uses two image recognition based applications with fixed datasets and models. The evaluation shows that when these models are deployed on the edge server, requests made from mobile devices experience lower response times, than when the model is deployed in the cloud.

Bonomi et al. [37] presents a more extensive computing paradigm called *fog computing*. Fog computing proposes to extend the idea of cloud computing all across the network between the cloud and the user, instead of only adding computational resources to last-mile infrastructure, thus bringing computational resources closer to the user and making the network smarter. These compute resources could potentially be used for running application software.

Both of these ideas propose the concept of "more computing at the edge", which will be necessary in realizing large-scale real-time recognition. Most proposed usage models for these edge resources rely on offloading, i.e. treating them like a mirror image of the cloud and choosing a fixed location where all the computation is done. However, these edge resources will not be as powerful as servers in the cloud and each edge server/cloudlet will potentially serve multiple users. Naive mirroring might overwhelm the edge servers. Thus, an efficient and effective usage model for edge resources is required.

1.2.4 Optimizing mobile-cloud computing for image recognition

Generic offloading approaches suggested above, for cloud or edge resources, do not take application properties into account. Following the proposal of these generic approaches, some research efforts have focused on optimizing recognition applications specifically. They assume that offloading is necessary for such applications and thus focus on using properties of image recognition to accelerate offloading-based approaches.

Glimpse [29] focuses on reducing bandwidth consumption and latency by dropping or filtering frames from the captured video stream, so that fewer frames are actually uploaded to the remote server for recognition. To filter frames, it proposes to select successive frames only when they are significantly different from the previous frame, and offload these frames. To make up for any network latency or compute latency at the server, it proposes to use the last known location of the recognized object and track it on the device while waiting for results.

OverLay [28] has a similar goal, but uses sensors present on mobile smartphones, such as accelerometers and gyroscopes, to predict which frames might be blurry and drops those frames, thus reducing bandwidth consumption. To reduce compute latency on the remote server, it depends on relative object positions. It first records an inter-object-distance map for a given physical area, and then uses it online to restrict the search space for recognition.

Both of these approaches primarily target reducing bandwidth consumption by filtering out low-quality or irrelevant frames from the video stream. Doing so also reduces the computational burden on remote services. Both of these are steps in the right direction to better support the increasing number of devices. However, as shown in OverLay's evaluation, it is still not enough. If more than one device running the application connects to the same remote server, latency increases significantly. Thus, a better approach for supporting large-scale real-time recognition is required.

1.2.5 Caching approach

A typical distributed caching framework reduces network latency by storing requested content in a hierarchy of caches. By having the lowest level of the cache respond to most of the requests, while the higher levels invoked only on cache misses, such a system reduces response times for content-related requests such as images, videos and web pages. The caching approach for recognition follows the same principle, in that the lowest level of the cache tries to respond to most of the requests, and the higher levels are invoked only on cache misses. However, to be able to serve the right information, the cache also needs to recognize the objects within the requests, since the requests are not identifiers of any kind. Thus, instead of only caching the content that needs to be served in responses to requests, a recognition cache also caches the trained model that is required for inference. This approach proposes to partition and distribute inference across the cloud, edge servers and devices, in a coordinated, cooperative and hierarchical manner, which entails distributing the trained model across the different levels.

This thesis proposes this approach, and presents a formulation for expected latency for such a cache-based system. We show that it is important to understand the effect of cache size on such a cache, and that the cache size needs to be taken into account when estimating the expected latency. The proposed formulation incorporates the effects of computation time and recognition accuracy for inference, along with network latency. Such an explicit formulation provides a mechanism to estimate expected latency.

This thesis then presents Cachier [39, 40, 41, 42, 43], a system which implements a distributed recognition cache, and uses the developed formulation along with other mechanisms to realize a scalable solution for real-time recognition. To address the problem of scale, Cachier leverages the opportunity provided by it. Cachier minimizes response times for applications running on multiple nearby devices *collectively*, instead of treating each device individually. Given that these applications recognize physical objects and things, users in the same vicinity will tend to have similar requests. For example, in a museum, multiple nearby users will seek information about the same paintings in the area they are in, while shoppers in a grocery store will likely seek nutritional information about packaged foods. Cachier leverages this spatiotemporal locality to allocate items in the distributed recognition cache and reduce the response time for requests from co-located devices.

Cachier's architecture can be seen in Figure 1.2. The components for the system are distributed across the cloud, edge server and devices. The cloud component is similar to a typical recognition service. Cachier's main contributions lie in the edge server and devices.

In the edge server, Cachier's implements a dynamic and latency-aware recognition cache. It leverages spatiotemporal locality along with the developed formulation, to decide which items to keep in the edge-cache and which to evict, to minimize response times.

To further reduce response time, Cachier efficiently uses resources available on the devices. It enables inference on devices by prefetching trained models onto the devices. It achieves this through collaboration between the devices and the edge server. The edge server predicts possible

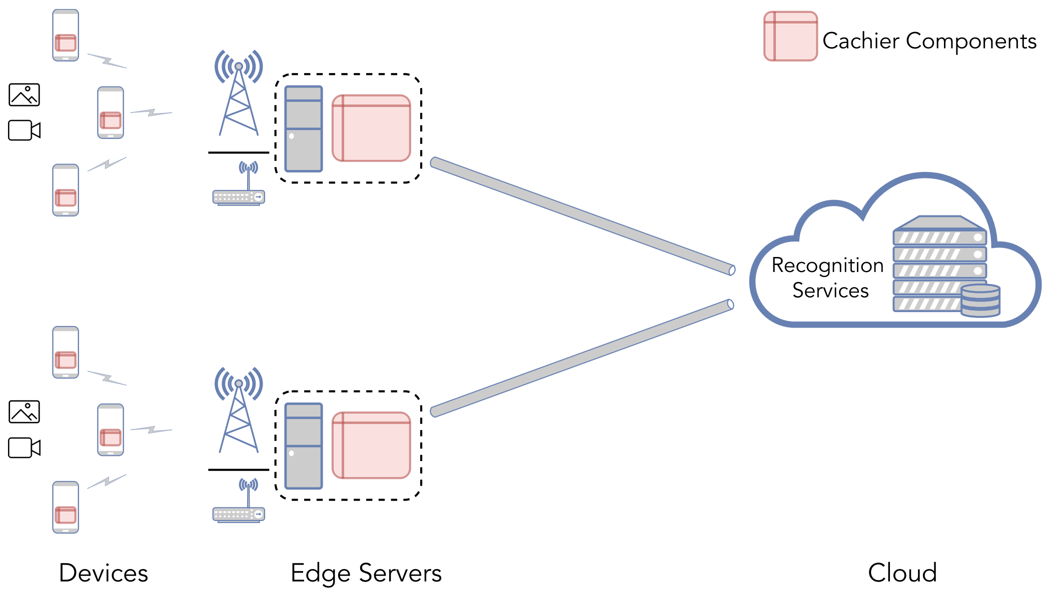


Figure 1.2: Cachier's distributed architecture.

future requests for each user, and the devices prefetch parts of the model accordingly. To further minimize latency of inference on the device, Cachier modulates the feature extraction phase for faster feature extraction when possible.

Finally, to leverage hyper-local context, Cachier enables coordination directly across nearby devices through a distributed key-value store hosted by the devices themselves. This allows devices to directly exchange information such as prefetched data and statistics about the visual environment, e.g. which features are currently prominent and easy to recognize. This can reduce the number of prefetch requests the edge server receives, and also enables the inference on the devices to potentially adapt to current characteristics of the immediate surrounding.

Chapter 2

Caching for recognition

Edge devices typically do not operate in isolation - in museums, stadiums, offices, streets etc., there are multiple users in close proximity who use them. If all of these devices run recognition applications, and try to offload their requests to remote services, that can be a cause for concern, since that may cause network bottlenecks, and will inundate remote servers with a deluge of data. This will increase latency as requests fight for network and compute resources. However, we note that recognition requests from applications on these devices will naturally be for similar objects, i.e. the requests will exhibit spatiotemporal locality across users. For example,

- In a museum, multiple nearby users may use applications like [44] to seek information about the paintings in that museum.
- In a grocery store, multiple shoppers may use an application like [45] to locate food items in that store that adhere to their dietary restrictions.
- In city streets, tourists and citizens may use an application like [46], or a wearable such as [9] to avail guided tours, locate restaurants or provide directions.
- Multiple autonomous cars on highways and roads will need to detect the same road signs or pedestrians on that road.

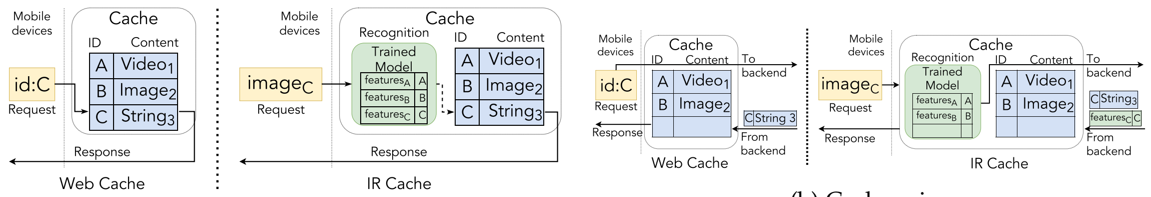
This thesis proposes that the challenge posed by the increasing number of edge devices can be addressed by leveraging this locality, and we propose to use recognition caches as the vehicle do so.

There are multiple reasons for using a caching-based approach. First, typically a cache's goal is to leverage locality of reference to make a popular subset of data accessible at a lower latency, which elegantly aligns with our goal. Second, a caching model comes with the benefits of a framework for reasoning about and analyzing a system, along with known knobs that can be tuned to optimize for different metrics. Finally, it also provides an inherent hierarchical structure that is extendable - caches can be added in a hierarchy. In the proposed mobile environment, the levels of the hierarchy can be seen as the device, edge-server and finally the cloud as the highest level. In this chapter we develop a formulation for the expected latency in such a cache, which will then be used as a guide when the cache is implemented on edge servers and devices.

2.1 Recognition Cache

Our approach has drawn inspiration from web caching and CDN systems. Web caching is an established technique of reducing user-perceived latency experienced when retrieving information from across the World Wide Web [47]. Caching web pages, objects and content in servers at the edge of the network reduces the time it takes for it to be delivered to users. The same idea is leveraged by content delivery networks [48]. Typically, these caches are not compute intensive - users' requests contain the specific identifier for the content they want to view and if the cache contains the content tied to the identifier, it is returned to the user, else the request is forwarded to the backend servers. In such systems, the cache has to maximize the amount of relevant content it can serve directly, and minimize the number of requests forwarded to the backend. This is how it can minimize the expected latency for users, reduce network load and backend load. A recognition cache also tries to minimize response times by minimizing the number of requests that are sent to the backend. However, it differs in how this is achieved.

A recognition cache would have the same high-level functionality of a typical cache, i.e. requests are intercepted by a cache and responses are sent back from the cache without going to the cloud . A recognition cache should have properties similar to that of a typical cache; it should be (1) transparent to the user, and (2) it should populate itself autonomously. However the two are not inter-changeable. The high-level operation of a cache that adheres to these properties is illustrated in Figure 2.1, alongside a web cache. The distinction becomes clear when we delve into how they operate.



(a) Cache hit

(b) Cache miss

Figure 2.1: Request look-up in a web cache and in an image-recognition cache. The incoming request for the web cache has a specific I.D. (C) for the content the user is looking for, which can be looked-up directly in the cache. For an recognition cache, the request is an image containing the object C. The recognition cache needs to first use recognition to know that the image contains C in it and then look-up C in the cache.

An image-recognition (IR) cache receives images, which themselves do not deterministically map to the content or information that needs to be returned to the user. The object within the image needs to be recognized, and then the object's identifier is used to return related content, unlike a typical cache, whose requests would contain the object's identifier itself. This is illustrated in Figure 2.1a. On a miss (Figure 2.1b), the cache forwards the request to the backend, receives the response, stores it locally and sends the response to the user. The image-recognition cache does the same, but along with the response, the backend also sends relevant parts of the trained model needed to recognize this request in the future. This is inserted into the model in the cache.

2.2 Formulating Expected Latency

The difference between a typical web cache and a recognition cache directly impacts the expected response time in a cache-based system.

To understand this difference in expected latency between the caches, we can adapt the AMAT (Average Memory Access Time) formula for caches. The AMAT formula is is given by $H + m * M$, where H = hit latency, M = miss latency and m = miss ratio. Adapting it for a networked cache-based system (Figure 2.2):

$$E[L] = L_{Cache} + m * (L_{Network} + L_P) \tag{2.1}$$

where L_{Cache} = Cache lookup latency, $L_{Network}$ = cache-to-parent network round-trip latency, i.e. sum of the time taken for a request to reach the parent after being issued from the cache and for the response to reach the cache after being issued from the parent, L_P = lookup latency at the

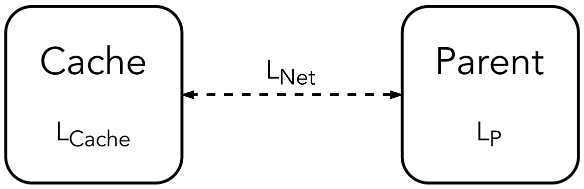


Figure 2.2: The expected latency in a cache-based system has to take into account multiple different sources of latencies, as shown here, and represented in Equation (2.1)

parent, and m = Miss ratio. Note, this formulation only considers time from the perspective of the cache, i.e. from the moment a lookup starts in the cache. It does not consider the time taken by a request to arrive at the cache, if suppose the client and the cache were not on the same entity.

2.2.1 Lookup latency

Lookup is trivial in web caches, e.g. similar to a hashtable lookup [48]. Lookup latency (L_{Cache}) is small, and the size of the cache has negligible impact on the lookup time. This is not the case in a recognition cache. Recognizing objects in the request image is compute-intensive, and lookup time is significant. Moreover, when an object is added to the cache, the model size increases, which directly impacts the lookup time [49, 50]. Thus, for a recognition cache, $L_{Cache} = f(k)$, $f(k)$ = function of cache size, k. Replacing in Equation (2.1),

$$E[L] = f(k) + m * (L_{Network} + L_P) \tag{2.2}$$

The lookup latency ($f(k)$) depends on the type of feature, the number of features extracted per object image and the classification algorithm utilized. For recognition algorithms, typically, $f(k)$ is a monotonically increasing function - as the number of recognizable objects increases, the time taken to process one input increases.

2.2.2 Miss ratio

For a typical web cache the miss ratio, m, depends on the cache size, the cache replacement policy and the underlying request distribution. A miss in such a cache happens only when the item being requested is not in the cache. However, there is another kind of miss in a recognition cache, a “recognition" miss. Given the nature of the lookup in a recognition cache, misses can happen even when the item being requested *is* in the cache. This can happen when the request image contains an object that is not recognized by the cache, even though the model in the cache is trained to recognize that object. The cache cannot distinguish between a “genuine" miss (object’s features not present in cached model) and a *recognition* miss. The request needs to be sent to the cloud in both cases. Such cache misses are bound to happen and will contribute towards overall latency. Hence, they need to be incorporated in the formulation of the expected latency.

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

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

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

Putting the formulation of miss ratio back in Equation (2.2),

$$E[L] = f(k) + (1 - recall(k) * P(cached)) * (L_{Network} + L_P) \tag{2.3}$$

Equation (2.3) formulation presents a detailed model of the expected latency in a recognition-cache-based system. We started with the high-level AMAT formula and extended it by adding the details for each term in the formula, to represent how they manifest in a recognition cache, and differentiate from a typical cache.

2.3 A Guiding Formulation

This formulation, Equation (2.3), reveals what underlying factors may affect the overall latency for recognition requests when using a recognition cache. At the same time, it also provides a mechanism to estimate what the latency might be. Cachier will use this, at the edge server and on the device as well, to estimate latency and also to make decisions that minimize the latency.

Chapter 3

Recognition Caching in Edge Servers

Typical systems for mobile recognition offload data directly to the cloud to run intensive algorithms. However, this is not feasible since it incurs high network latency, which will only get worse as more devices become users of the Internet. To offset this communication latency, it has been proposed that the computation be brought closer to the devices by putting compute resources (servers) at the edge of the network [37]. These edge servers are placed near the users and are capable of serving requests in a similar manner as the cloud, thus avoiding the edge-to-cloud latency [36]. The state-of-the-art framework to use edge servers is similar to how applications use the cloud, i.e. offload intensive tasks to the edge servers. However, edge servers will not have as much compute resources as the cloud. If edge servers are used as a drop-in replacement for the cloud, they are likely to be overwhelmed. Hence a better approach, which adapts to the available resources at the edge, is required.

We propose to use these edge servers as recognition caches. Essentially, instead of deciding which parts of the application to execute at the edge, we propose to dynamically decide which subset of the recognition classes to be included in the trained model at the edge. We will use the formulation developed in Chapter 2 to make this decision.

3.1 Applying the Recognition Cache Model

A recognition edge cache is an application of the abstract model that was developed earlier in Chapter 2. Here the cache is present at the edge server and the parent is the recognition service in the cloud, as shown in Figure 3.1. Recognition requests from devices in the vicinity of the edge

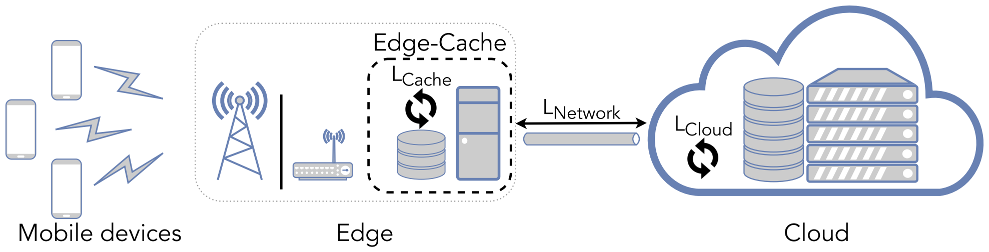


Figure 3.1: System architecture of a recognition edge cache backed by the cloud.

server are received by the recognition edge cache. If the edge cache successfully recognizes the required object, it returns the response. Else the request is forwarded to the cloud. The cloud's response to edge cache contains both, the response and the part of the trained model that was used to recognize the object. The edge cache returns the response to the requesting device and updates its own cached model.

We can rewrite the expected latency formulation using relevant terms:

$$E[L_E] = f_E(k) + (1 - recall_E(k) * P_E(cached)) * (L_{Network} + L_{Cloud}) \tag{3.1}$$

where the subscript E represents Edge.

3.2 Implications of Cache Size

As we mentioned earlier, edge servers will not have as much compute resources as the cloud and hence cannot be used as a cloud substitute. This also applies when using a recognition cache at the edge. The cache needs to be resource-aware.

To investigate further, we evaluate a recognition edge cache with increasing cache sizes, set up as shown in Figure 3.1. Figure 3.2 shows the results for that experiment. Note, increasing the cache size increases the cached model's size. In this experiment, the cloud is a resourceful server (12 cores, 24 Hyper-Threads, 32GB RAM) while the edge-server is a powerful PC (4 cores, 8 Hyper-Threads, 8GB RAM). The request distribution is a Zipf distribution with $\alpha = 0.8$. We chose the Zipf distribution since user requests over the WWW tend to have a Zipf distribution [51]. The network conditions between the edge and cloud is fixed at 5 Mpbs and 40ms RTT. The cloud contains a trained model for all the objects that can be present in images (400 objects). Before

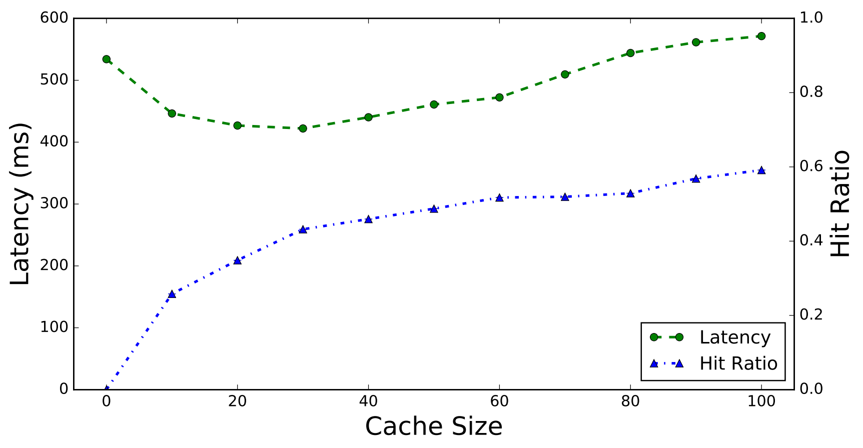


Figure 3.2: Overall latency and hit ratio for a recognition edge cache, over different cache sizes.

the measurements are taken, the cache is warmed to be full with the most frequent items, thus emulating an LFU-based cache.

Initially, as the cache size (k) increases, the hit ratio increases, i.e. the cache serves more items locally and avoids sending requests to the cloud, and the overall latency decreases. Although the cloud is computationally powerful, it needs to classify incoming images as one of 400 known objects. The cache, on the other hand, knows of k objects. Hence it tries to classify incoming images as one of k objects, where $k < 400$. This lowered computational load, combined with the effects of the network conditions, favors having a cache.

However, this is only true up to a point. For cache sizes over 30, the latency starts to increase. In fact, for cache sizes 80 and above, the latency is worse than when there is no cache at all (cache size is 0). This is counter-intuitive - for a typical cache, a bigger size implies lower latency. For a recognition cache, however, there are other factors in play. The size at which the inflection occurs is affected by multiple factors - the nature of $f(k)$, the request distribution, the network and cloud conditions ($L_{Network}$ and L_{Cloud}). As the size increases, $f(k)$ starts having a strong effect. Since the computational resources of the edge server are lesser than the cloud, the processing overhead is higher than the impact of sending requests to the cloud. This slows down the entire system because $f(k)$ is unavoidable - each request will at least take that much time, whether it hits or misses in the cache. Thus a simple, static LFU cache cannot be used for image recognition applications, since it does not incorporate the effects of all these factors. An adaptive approach is required that can decide which objects to cache based on these different factors.

3.3 Minimizing Response Time

The edge cache needs to decide how to allocate the cache such that it can respond quickly to requests from the users, i.e. the average response time needs to be minimized. This average response time is represented by Equation (3.1). Hence our approach to minimize response times will be to use this expected latency formulation and find the k, i.e. cache size, that minimizes this formulation. As we see in Equation (3.1), there are four terms that make up the formulation, (1) $f_E(k)$, the compute time of matching a request to a known cached object, (2) $recall_E(k)$, the probability that a request is correctly recognized given the corresponding object is in the cache, (3) $P_E(cached)$, the probability that a randomly chosen object is in the cache, and (4) $L_{Network} + L_{Cloud}$, the sum of the network RTT and cloud-lookup time. By estimating or measuring each term of the formulation, the expected latency can be estimated and then the optimization can be carried out. We look at each in the above order and show how we can estimate these.

3.3.1 Offline Estimation of $f(k)$ and $recall(k)$

The effect that changing the cache size would have on the latency and accuracy of the recognition algorithms at run time needs to be estimated. The compute time and accuracy of the recognition algorithms depend on the choice of the algorithm, the kind of objects they are supposed to identify, i.e. the dataset, and the size of the trained model, that is the number of objects the model is trained to recognize. For a given application, either the application developer decides the recognition algorithm and dataset, or it is left upon the recognition service being used by the application. In either case, these are known before hand and do not change at run time. The size of the trained cached model is the cache size, which is modulated at run time. Hence, essentially, the $f(k)$ (latency) and $recall(k)$ (accuracy) need to be estimated under different model sizes.

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

3.3.1.1 Procedure

First, the list of object classes in the dataset is sampled to select k object classes. A model is then trained offline on the training data for the selected classes. This trained model is then used as the cached model and requests are made to this cache. The compute time, recall and precision for looking up request images is measured. The image requests consist of test images of objects that are known to be in the cache (relevant requests) and test images of objects that are known not to be in the cache (noisy requests). For each cache size, 8 different random subsets of the dataset are taken for rigor. This is carried out for a range of values of k spanning the given dataset, i.e. from 0 to total number of object classes in the dataset. The collected compute time data and recall data for each value of k is then used to estimate $f(k)$ and $recall(k)$ respectively. The estimation of these polynomials is done through regression analysis.

3.3.1.2 Setup

We estimate $f(k)$ and $recall(k)$ for two algorithms, the basic brute force algorithm (BF) and multi-probe Locality Sensitive Hashing (LSH) [52]. The trained models created by these algorithms are quite different.

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

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

The binary ORB [23] features are extracted and used for the recognition.

We estimate the $f(k)$ and $recall(k)$ for these algorithms on two datasets, which we call Stanford [53] and UMiss [54], each consisting of 400 objects like paintings, CD covers, book covers, movie posters, magazines, and food packaging. Each dataset has one perfect training image for each object and multiple test images taken from mobile devices. Some examples of training and test images are shown in Figure 3.3. The models based on features from the training images are the ones that are first stored in the cloud and then in the caches, while the test images are used as the request images sent from the mobile device.

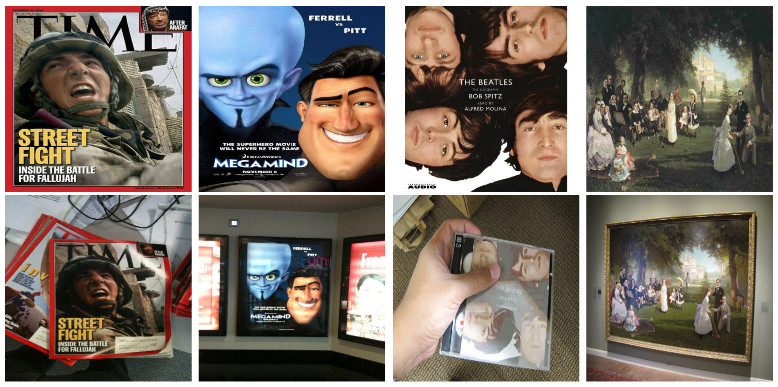
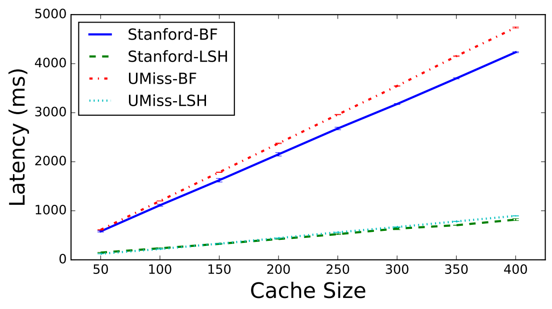
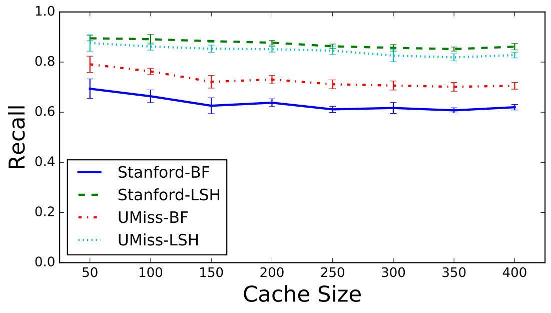


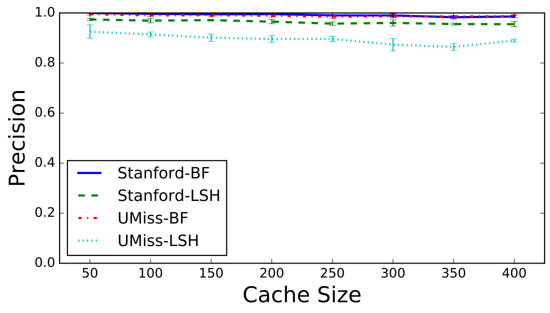
Figure 3.3: **Top:** Training images from the dataset. **Bottom:** Corresponding query images in dataset, taken by mobile devices.



(a) Effect on latency - $f(k)$



(b) Effect on recall - $recall(k)$



(c) Effect on precision

Figure 3.4: Offline Analysis of two datasets, using two algorithms. Note that high precision is important to maintain accuracy.

The mean and standard deviation of the measurements using these two algorithms on the two datasets are recorded for each cache size in Figure 3.4. The estimated polynomials are presented in Table 3.1 and Table 3.2. The presented estimates are linear because higher order terms are insignificant, which is apparent from the measurements in Figure 3.4 as well.

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

Table 3.1: Estimated $recall(k)$

3.3.1.3 Precision and Recall

Recall was calculated as the fraction of true requests that were correctly answered, while precision was calculated as the fraction of definite responses (not “unknown") that were correct. Although both high recall and high precision are desirable, high precision is especially important. It is required that responses that go back to the devices directly from the cache should have a high probability of being correctly recognized. The presence of a cache should not negatively affect the accuracy. This is achieved when precision is high, i.e. the cache correctly recognizes true requests with high probability.

We can see in Figure 3.4b that the recall dips slightly with increase in cache size and the trend is similar across the datasets. On the other hand, precision is constantly high (>90%) across cache sizes. The estimated polynomial for $recall(k)$ for the different algorithms and datasets are listed in Table 3.1.

3.3.1.4 Analyzing f(k)

We see that LSH is certainly much faster than brute force. The trend is very similar for both datasets (Figure 3.4a), as the cache size (k), that is the number of trained objects in the model, increases, $f(k)$ increases. LSH is the default algorithm used in Cachier. The estimated polynomial for $f(k)$ for the different algorithms and datasets are listed in Table 3.2.

Note, the trade-offs between latency, recall and cache size seen here are specific to the algorithms being used. The same trade-offs may not hold for a different set of algorithms, or if suppose no geometrical verification is involved. It may happen that the latency grows at $O(nlogn)$ or $O(logn)$ and not simply linearly. However, the approach of estimating the relationship, and using it online will remain the same. What will change is the degree of the polynomial used for the regression analysis.

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

Table 3.2: Estimated $f(k)$

3.3.2 Estimating $P(cached)$

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

3.3.2.1 Leveraging spatiotemporal locality

An edge server serves a small physical region, dictated by the network element it is co-located with. This implies that the requests arriving at the edge-server are from users in the same physical region, e.g. at the same coffee shop, same grocery store or same neighborhood. In such scenarios, users' recognition requests from applications will naturally be for similar objects, i.e. the requests will exhibit spatiotemporal locality across users. For example, in a museum, multiple nearby users will seek information about the same paintings in the area they are in. Our approach will leverage this locality to cache relevant objects at the edge, recognize them there and minimize the number of requests that go to the cloud. To leverage this spatiotemporal locality, we start with the Least Frequently Used (LFU) policy and modify it to create a cache-inclusion policy instead of an eviction policy. LFU eviction policy evicts the least frequently used item in the cache when the cache is full. Conversely, this policy keeps the most frequently occurring requests in the cache. Now suppose that the cache size is one, $k = 1$, then the cache only has the most popular object, and hence the probability of a cache hit is the probability of a request being for the most popular object. Let us now extend this notion. Let i denote the rank of popularity of an object, $i = 1, 2, ..., N$, 1 being the most popular. Let $request_i$ denote a request for an object with popularity rank i. We can then say:

If $k = 1$, then $P(cached) = P(request_1)$

If $k = 2$, then $P(cached) = P(request_1) + P(request_2)$

If $k = n$, then $P(cached) = \sum_{i=1}^{i=n} P(request_j)$

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

3.3.2.2 Estimating request distribution

The request distribution is modeled as a multinomial probability distribution, $P(request_i) = p_i$, $i = 1, 2, ...N$, and $\sum p_i = 1$, where N is the total number of object classes, and i is ordered in decreasing order of probability. Thus, $P(cached) = \sum_{i=1}^{i=k} p_i$, for a cache of size k.

Such a distribution can be estimated using MAP (maximum a posteriori) estimation. This requires the system to track the number of times each object i is requested, which is represented by M_i. Using this, the estimate is given by,

$$\widehat{p_i} = \frac{M_i + \alpha_i}{\Sigma M_i + \Sigma \alpha_i} \tag{3.2}$$

α_i is the Dirichlet prior, which is used to provide a prior to avoid overfitting to incoming data, especially when enough data has not been collected. This is generally set to 1, to convey that all objects are uniformly distributed. However, α_i can be used to insert different priors if needed, and we discuss this in Chapter 3.5.

Thus, finally, for a cache of size k,

$$P(cached) = \sum_{i=1}^{i=k} \frac{M_i + \alpha_i}{\Sigma M_i + \Sigma \alpha_i} \tag{3.3}$$

3.3.3 Cloud and Network Profiling

The only remaining component in Equation (3.1) is the latency penalty on a miss, namely $L_{Network} + L_{Cloud}$. Cachier tracks this on the edge server and keeps a moving average filter to even out noise. For every miss in the cache, the time between issuing the request to the cloud and receiving a response from it is measured. This essentially measures the sum $(L_{Network} + L_{Cloud})$ as a whole instead of each term individually. This is useful because if the cloud is overloaded with requests and is taking longer to compute responses then this gets incorporated in the measurements and allows Cachier to adapt dynamically.

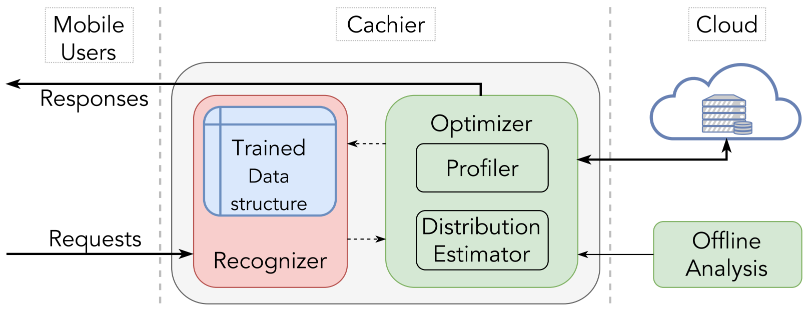


Figure 3.5: Internal architecture of recognition edge.

Now we have a way to estimate the effects of changing the cache size on each of the components in the Equation (3.1). Replacing the $P(cached)$ term:

$$E[L_E] = f_E(k) + (1 - recall_E(k) * \sum_{i=1}^{i=k} \frac{M_i + \alpha_i}{\Sigma M_i + \Sigma \alpha_i}) * (L_{Network} + L_{Cloud}) \tag{3.4}$$

This is the final formulation that needs to the estimated. All terms in the formulation are now either functions of the cache size or independent of it.

3.4 Adaptive Recognition Cache

We can now design a system that leverages Equation (3.4) to predict expected latency for different k by measuring and estimating the different unknowns in this formulation, using the described methods. Figure 3.5 shows the internal architecture of this system. It contains different modules that work in concert to dynamically estimate latencies for different cache sizes and choose the one that minimizes latency.

The Optimizer module brings these techniques together to find a cache size that minimizes the expected latency. It estimates the formulation by using the estimated $f(k)$ and $recall(k)$ polynomials from the Offline Analysis, the estimated request distribution to find $P(cached)$ from the Distribution Estimator, and the measured miss penalty from the Profiler.

The Distribution Estimator carries out the computation to estimate the request distribution and hence $P(cached)$ as shown in Figure 3.5. On receiving a request, the object in the request is first recognized, either by the Recognizer, using the local model, or, on a miss, by the cloud. Once the request is matched to a known object, the Estimator updates the respective object's counter.

When a $P(cached)$ estimate is needed for optimization, it calculates the probabilities using the counters and MAP estimation.

The Profiler measures the response time every time a cache miss occurs, thus estimating the sum $(L_{Network} + L_{Cloud})$.

Periodically, the Optimizer searches for a cache size, k, that minimizes the formulation given in Equation (3.4) using gradient descent. This decides dynamically how many of the most frequently occurring requests should be included in the cache. Once it finds the right cache size k, it places the top k items in the cache, that is the model in the cache can now recognize those top k objects. The top k items is known from the distribution estimated by the Distribution Estimator. The Recognizer then uses this updated model to execute future recognition requests. Given the simplicity of the function, it is not expensive to compute the expected latency for a given k and it takes less than 1ms.

3.5 Optimizations

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

3.5.1 Multi-user Support

When there are multiple users in the same vicinity, making simultaneous requests to the edge server then other latencies come into affect which need to be addressed. Requests will start queuing up if the compute time of a single request is higher than the average inter-arrival time of requests from the devices. At the same time, cloud resources will be under utilized because all the requests will be held up by the edge cache. To ensure that requests do not queue up, two techniques can be employed; (1) the compute time needs to adapt to the inter-arrival time, i.e. it has to always stay below the prevailing inter-arrival time, and/or (2) requests should be offloaded to the cloud when the edge server cannot serve all incoming requests.

3.5.1.1 Bounding compute-time

The compute time on the edge server is decided by the cache lookup time, i.e. by $f(k)$. To ensure that requests are not queued at the edge server, $f(k)$ needs to be lesser than the average inter-arrival time of requests. We already have a knob that can control this - the cache size k. Thus to avoid requests from queuing up at the edge server, k needs to be set in a manner that $f(k)$ is lower than the inter-arrival time. At the same time, k is also set by minimizing the expected latency cost function Equation (3.4). Here, we leverage the monotonically increasing nature of $f(k)$. Working backwards from the inter-arrival time, the search space for k can be restricted between 0 and an upper-bound.

Estimating inter-arrival time. To keep track of the inter-arrival time, Cachier pushes the inter-arrival time for each request into a rolling buffer and uses the mean of the buffer's contents when the average inter-arrival time is required.

With an estimate of the mean inter-arrival time, IAT_{mean}, we set $f(k) = IAT_{mean}$ and find the corresponding k and apply the floor function to it to get an integer, k_{max}. This k_{max} represents the k up to which the search for a suitable cache size should extend. Going above this k will increase $f(k)$ (due to the monotonic nature) and thus go over IAT_{mean}.

Automatic load-balancing. By reducing the cache size when the inter-arrival time is low, more requests are automatically forwarded to the cloud since requests start missing in the edge cache. This ensures that under high loads, cloud resources are also utilized. By keeping the most popular objects in the cache, the edge server can still manage to respond to a high number of requests.

3.5.1.2 Explicit Load-balancing

The other approach is a more direct, cache-agnostic way of handling multiple simultaneous requests. Given that there are resources available in the cloud, we can treat the cloud simply as another server to which requests can be directed to. In this approach, the edge server is treated as the primary server, and the cloud as a backup, similar to the caching hierarchy. However, instead of forwarding requests to the cloud after going through the cache, requests are redirected to the cloud before being looked up in the cache. To do this, we create a load balancer module as the first entity the requests enter as they reach the edge server.

Load balancer. This module probabilistically allocates each request to the edge server or the cloud. To make this decision, it uses the average inter-arrival time of requests, and the average lookup time in the cache. Both these quantities are estimated as in the previous approach. The module takes checks the ratio of the average inter-arrival time to the compute time. If this quantity is greater than one, it means that the edge server is fast enough to compute the request, and hence the request is allocated to the edge server. If the ratio is less than one, then the edge server certainly cannot keep up with all the requests and needs to offload. Then the probability of the edge server processing the request is set to the calculated ratio. This ensures that for a large number of requests, the edge server handles as many requests as it is capable of, and offloads the rest to the cloud.

3.5.2 Learning from Previous Operations

If Cachier is powered up for the first time, it has no historical data and starts from scratch, with uniform Dirichlet priors for the request distribution. However, once operation starts, Cachier periodically logs the request distribution to disk, and the next time it boots up, it can use this history as the priors for the estimation. The priors are inserted as the α when finding the MAP estimates of the request distribution (Chapter 3.3.2). This lets it predict the stable cache size right after startup, without waiting to collect data from requests. This should be used when the request distribution does not change significantly.

3.5.3 Lazy Model-fetching

Originally, the request is forwarded to the backend in the cloud on each cache miss and the reply contains both the response for the user and the relevant part for the model in the recognition cache. A recognition cache miss occurs if the cache doesn't recognize the object in the request, but it still might have the relevant part in its model. If this part of the trained model is fetched from the cloud in such cases, communication bandwidth will be used needlessly. To optimize this interaction in Cachier, the backend only returns the object identifier and the response to Cachier. Cachier checks if the identifier is present in its local trained model. Only if it is not, it requests the respective part using the identifier returned by the backend. This eliminates the unnecessary traffic between the edge and the backend.

3.5.4 Model Request Holdback Queue

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

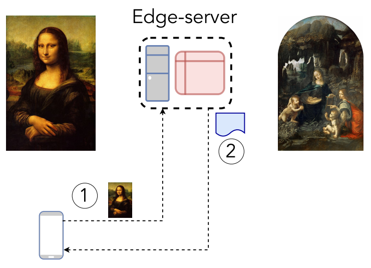
Chapter 4

Prefetching for On-device Recognition

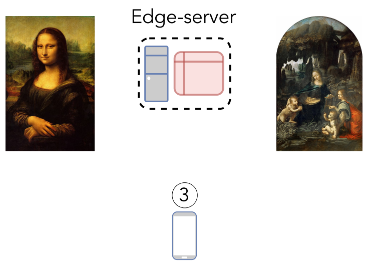
As introduced in the previous chapter, one approach to reducing the impact of network conditions on cloud-based mobile recognition applications is edge computing. Edge computing places compute resources at the edge of the network, and hence closer to the user. By offloading recognition tasks to these "edge servers", applications can avoid the latency incurred when offloading to the cloud. The recognition edge cache takes a caching approach and uses these edge servers as caches. It also takes into account the difference between available resources at the edge and the cloud. These techniques essentially lower the edge-to-cloud interactions and hence lower the recognition latency. However, this does not affect the last-mile network latency, that is the latency due to the wireless communication between devices and the edge. Moreover, with the rapid increase in the number of users using these applications, the edge server itself can potentially become a bottleneck and lead to higher recognition latency.

The logical extension to address this issue is to push the computation even closer to the devices, that is the devices themselves should do the computation required for recognition. This would avoid latency due to communication and would not use any remote resources. However, the primary motivation of offloading in the first place was that edge devices do not have sufficient resources to power recognition applications, and that it was in fact faster to offload tasks to remote servers at the edge or in the cloud.

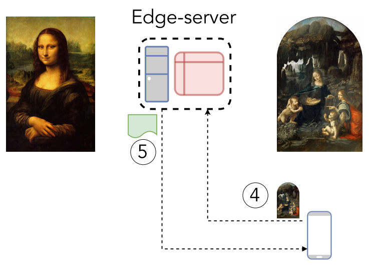
We propose to use the resources available on edge devices efficiently by collaborating with edge servers, leveraging locality and selectively recognizing images on the device. We propose a system based on prefetching for image recognition applications, which employs an on-device



(a) The user uses the device to capture an image, which sends it as a request to the edge server, the edge server recognizes the painting and sends the related information back to the device.



(b) The user moves on to the next exhibit.



(c) The same process as (a) takes place; upload of the request, recognition at the edge server and then reception of the related information.

Figure 4.1: This figure represents the scenario in which a visitor in a museum is using a mobile device to recognize paintings to know more about them. Here the system does *not* use prefetching.

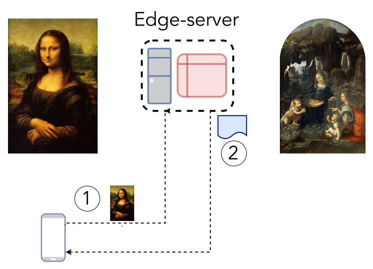
recognition cache and uses prefetching to avoid compulsory misses, while minimizing computation on the device.

4.1 Prefetching for Recognition

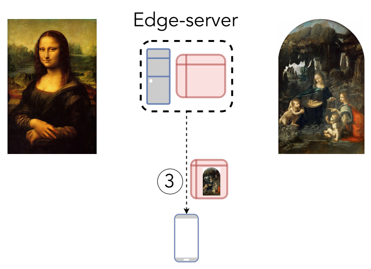
A recognition cache in the edge server caches relevant parts of the trained models that the recognition algorithms use to recognize objects. Since edge servers serve a restricted physical area, dictated by the networking element they are attached to, it is likely that the requests they receive have high spatiotemporal locality. For example, an edge server might receive requests from shoppers in a grocery store, or visitors in a museum. The recognition edge cache leverages the spatiotemporal locality in requests from *across* the devices. If it sees a request for an object for the first time, a cache miss occurs, and the required data is fetched from the cloud. Then requests for the same object from *all* devices can result in a cache hit. In this manner, a recognition cache in the edge server minimizes expected latency across devices, and amortizes the cost of the first cache miss. By storing only parts of the model, the cache can accelerate classification of incoming request images.

This idea could be extended to devices by employing similar recognition caches on the devices themselves. However, since the device cache will only see requests from the individual user, using only a caching technique will lead to a high number of compulsory misses.

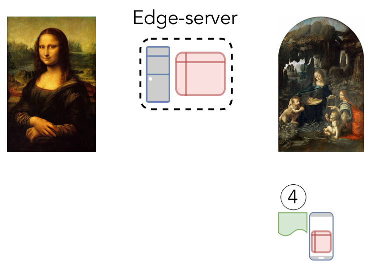
To elaborate, let us consider a scenario in which a visitor is walking through a museum while



(a) The user uses the device to capture an image. Since the device does not have a model for this painting, it sends it as a request to the edge server, the edge server recognizes the painting and sends the related information back to the device.



(b) While the user moves on to the next exhibit, the edge server predicts what the user might see next. It sends the model for the next painting to the device.



(c) The user takes an image again, but this time the device does have a model for this painting. It recognizes the painting locally, and serves the related information to the user, thus avoiding going to the edge server and minimizing response time.

Figure 4.2: This figure represents the scenario in which a visitor in a museum is using a mobile device to recognize paintings to know more about them. Here the system uses prefetching.

using a mobile application that recognizes exhibits and provides additional information, as shown in Figure 4.1. As the visitor moves among exhibits, she will come upon new exhibits. Every time a new exhibit is encountered, it will not be recognized by the device cache and will lead to a cache miss. Such a cache miss is called a compulsory miss. These misses will raise the expected latency and will not be amortized in the same manner as in the edge server.

We propose to avoid these misses by prefetching the required items into the cache *before* they are requested, as shown in Figure 4.2. If the edge server knows what the user might see next, it can provide the model to the device, so that the device itself can carry out the recognition process. This will minimize the number of requests that result in cache misses on the device and thus reduce latency due to the network and the remote computation.

4.1.1 Prefetching

The concept of prefetching originated in computer architecture literature, and has consequently been applied in distributed systems to reduce user-perceived latency when loading pages from the World Wide Web (WWW), across the Internet [55]. The concept behind prefetching here is to leverage the fact that most web pages link to other web pages. The web server can keep track of which web-page link visitors tend to visit next from a given page. The server can use this information and push the next probable web pages to the client machine. Now, if the visitor clicks on a link that the server determined probable, it is readily available on the client machine, and the visitor does not need to wait for it to be downloaded from the server *after* the link is clicked. Essentially, the download has been moved from the critical path and instead takes place in the background, such that the user need not experience that delay. We propose to use the concept of prefetching, but apply it to mobile image recognition applications.

Prefetching can be challenging on already-resource-constrained devices because it can lead to additional, potentially wasteful, computation. We design a recognition cache for devices, enhanced with prefetching, to use the following techniques to meet the challenges that arise due to resource constraints.

4.1.1.1 Device-Edge Collaboration

Since the edge server sees requests from a number of devices in a given physical area, it can obtain statistics about objects in that area. By monitoring these requests from across users, Cachier estimates a Markov model that describes the relationship among the objects in that area that users are querying. This model is then used to make predictions about what a user might query in the future.

4.1.1.2 Expected latency minimization

We extend the expected-latency formulation, which incorporates the network conditions, the device capabilities and the predictions made by the edge server. On each device, Cachier minimizes this formulation by choosing an appropriate part of the trained model that should be cached and used for recognition locally on the device.

4.1.1.3 Adaptive feature extraction

The compute time of recognizing an image includes both feature extraction and classification, as seen in Chapter 1. Since the cached model on the device is trained to recognize few objects, we find that the classification of the request image's features is typically fast. Instead feature extraction can be the bottleneck on the devices in such cases. We develop a technique that dynamically reconfigures feature extraction, based on multiple runtime conditions, to minimize overall latency.

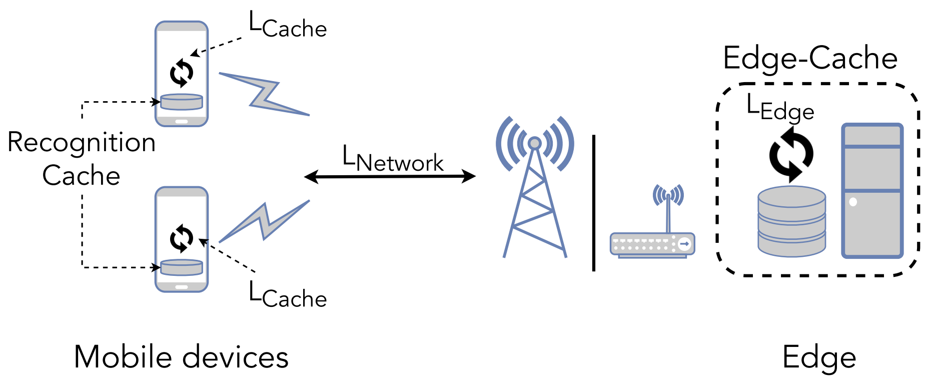


Figure 4.3: System architecture of devices with recognition caches, backed by the edge cache in the edge server.

4.1.1.4 Optimized Model Loading

Prefetching will require constant updates to the cached trained model on the devices. Every time a new object is encountered, the model on the device will need to be changed to contain the next possible set of objects. This will lead to models being constantly loaded into memory. We design optimizations that ease this stress on these resource constrained devices by using immutable data structures and caches for the models.

4.2 Applying Prefetching to the Cache Model

The cache on the device is still a recognition cache. However, instead of deciding what to evict from the cache, the device cache needs to decide what to prefetch into the cache. Once items are stored in the device cache, the mechanism of lookup is the same as that for a recognition cache. As shown in Figure 4.3, it is backed by the edge cache in the server, i.e. requests go to the edge cache when they miss on the device cache. This implies that the formulation for expected latency from the device's perspective remains the same as the general formulation derived in earlier chapters. Rewriting the expected latency formulation for prefetching on devices,

$$E[L_D] = f_D(k) + (1 - recall_D(k) * P_D(cached)) * (L_{Network} + L_{Edge}) \tag{4.1}$$

Where, the subscript D denotes that the term is from the perspective of the mobile device and L_{Edge} = is the lookup latency in the edge server. This will be the expected latency for future lookups if a model of size k is prefetched on the device and used for the lookup. The future expected latency can be estimated by estimating each term in this formulation in terms of k. Then by finding the k that minimizes it, choosing which objects make up those k objects and prefetching the model for those objects, the device can minimize $E[L_D]$, end to end latency or recognition latency from the device's perspective. The challenge will again be to estimate the different terms in the formulation, in the context of prefetching on a resource constrained device, such that the formulation can be minimized to find which parts of the trained model should be prefetched and stored on the device.

4.3 Device-Edge Collaboration

To minimize the expected latency, the device cache will need to decide *what* should be prefetched, along with determining how many objects (k) need to be prefetched. For an efficient cache, these objects should be the most likely objects that the user will request in the immediate future. This needs to be captured in $P_D(cached)$. As mentioned earlier, this term is the probability of a hit in the cache. The higher this term is, the lower the expected latency will be, when other terms are constant. A high $P_D(cached)$ reflects the belief that the items put into the cache are the ones the user is most likely looking for, i.e. the probability of a hit is high, *in the future*. This is challenging to estimate from only the individual device's perspective. An external source is required for this.

The edge cache sees requests from a number of devices in its vicinity. It is in a good position to estimate the requests the user may make in the future, based on a user's current request. We propose to estimate a Markov-model from the requests that the edge server receives, and then use that model to predict what a user might see next, given her current query. Instead of only providing the most likely next object, the edge server will provide a probability distribution over all possible next objects. This will be used by the devices to calculate $P_D(cached)$.

4.3.1 Edge-mediated Markov Prediction

The edge server has to serve the probability distribution over next possible objects, PDF_j, to each device that is connected to it, based on the last object the user has queried about. This will be used to calculate $P_D(cached)$, an integral component in the latency formulation Equation (4.1). The edge server estimates $P(next)$ by first estimating a Markov model from the requests it receives from the devices.

4.3.1.1 Markov model

Given a set of states, a Markov model describes the probabilities of transitioning from one state to another. The Markov assumption is that these probabilities are dependent only on a finite history of the previously visited states. A first order Markov model implies that the probability of transitioning from one state to another state only depends on the current state, and not on any other states that have been visited in the past. Cachier estimates and uses a first order Markov model at the edge server.

In our approach, these states are the object classes to be recognized themselves. For example, in a museum each exhibit would be a state in the model, or in a grocery store, each recognizable product would be a state in the model. Hence the model essentially captures how users transition from object to object. Then, by knowing what a user queried last, it will be possible to predict the next object that the user might query about. Although typically a Markov model is used to predict the next most likely state (or object in our case), in our approach we use the entire probability distribution of the possible next objects. We elaborate on this later in the section.

4.3.1.2 Estimating the Markov model

Estimating the model essentially means estimating the transition probabilities. The transition probabilities themselves are conditional probabilities, which can be depicted by $P(S_j|S_i)$, where S_i is the current state and S_j is the next possible state. A Markov model can be estimated by estimating these conditional probabilities for all i, j. Cachier uses MAP (maximum a posteriori) estimation to estimate each conditional probability. This estimate is given by $\widehat{P}(S_j|S_i) = \frac{N_{i,j}+\alpha_{i,j}}{\Sigma_j N_{i,j}+\Sigma_j \alpha_{i,j}}$, where $N_{i,j}$ is the total number of transitions from S_i to S_j, and $\alpha_{i,j}$ is the prior. A prior is important during bootstrap - it minimizes the chances of overfitting when not enough data is available.

Note that the counts, $N_{i,j}$, are independent of the user or device. They are counts for whenever a user requested for object S_j after requesting for object S_i. To track these counts on the edge server, Cachier tracks the requests from each device separately but updates a global data structure which stores the counts. For every device connected to edge server, Cachier remembers the object that was recognized in the last request it has seen from that user, say S_i. When it sees a new request from the same user, which contains the object S_j, it increments $N_{i,j}$ in the global data structure. Hence, Cachier learns a single Markov model in a given edge server, which captures the prevailing flow of users between objects that are in the area served by that edge server.

4.3.1.3 Using the Markov model

The edge server uses the Markov model to provide PDF_j to the devices. PDF_j is not a single value. Instead, it is the probability distribution over all possible objects that the user might query for next, after querying for object S_i. The edge server can provide this to the devices in two ways.

Piggyback on response. In the first method, the PDF_j is piggybacked on responses to requests made by the device. If a request results in a miss in the device's cache, it is received by the edge server. After recognizing the object in the request, the edge server knows the current object (S_i) the user is looking at. The edge server then looks up all the possible next objects, (all S_j) with non-zero probabilities of transition in the estimated Markov model. It then orders these in descending order of the probabilities. This forms the PDF_j. It appends this list to the response being sent back to the device. Note, the list only contains object IDs and their corresponding probabilities, not the models for those objects.

Explicit request. In the second method, the device explicitly requests for the PDF_j and provides the current object (S_i) in the request. This is utilized when the recognition happens on the device itself and the device needs PDF_j to decide what it should prefetch next.

Note, in both cases $PDF_j = [< S_1, P(S_1|S_i) >, < S_2, P(S_2|S_i) >, ..., < S_M, P(S_M|S_i) >]$, where S_i is the last requested object, and there are M objects with non-zero conditional probabilities. Once PDF_j is received by the devices, each device uses it when estimating the k that minimizes the expected latency.

The size of the message carrying predictions from the edge server to the mobile device is linear in the number of objects in the edge server. However the process of sending predictions to the device is in the background and does not directly affect the latency from the device's perspective.

4.4 Estimating Expected Latency

Equation (4.1) can be used to estimate the recognition latency the device might experience, and help decide what should be prefetched to the device. This requires estimating $f_D(k)$, $recall_D(k)$, $P_D(cached)$, and $(L_{Net} + L_{Edge})$.

4.4.1 $f_D(k)$ and $recall_D(k)$

$f_D(k)$ represents the time it takes to recognize an object in an image given that the model being used for recognition is trained to recognize k objects. This term is crucial since it estimates the hit latency in the cache. It is also the constant penalty for using a cache - irrespective of hit or miss, $f_D(k)$ must be spent for lookup. $recall_D(k)$ represents the accuracy of the recognition algorithm. A request is considered an accuracy-based cache miss if it is not recognized, even though the cached model is trained to do so. By incorporating $recall_D(k)$ into the latency formulation, we ensure that such misses are accounted for since those misses will contribute towards latency as well. Both of these terms, $f_D(k)$ and $recall_D(k)$, depend on, apart from k, the algorithm being used for recognition and the compute resources available on the device. Since these dependencies do not change at runtime, $f_D(k)$ and $recall_D(k)$ can be estimated offline and be used online.

To estimate these terms offline, a number of runs of the recognition application are carried out on the device. Each such run generates points for recognition latency and recall. This is done for multiple different sizes of the model on the device by changing k, and the mean and standard deviation for recognition latency and recall for that k are recorded. This recorded data is then used to estimate polynomial functions for $f_D(k)$ and $recall_D(k)$ using regression analysis. This process is very similar to the one described in Section 3.3.1.

Note, the $f_D(k)$ captures the recognition time as a whole. Internally, it includes both the extraction time and the classification time. We delve into this in Section 4.5 and show how this can be further optimized for device caches.

4.4.2 $P_D(cached)$

As we discusses in Section 4.3.1, $P_D(cached)$ is calculated from PDF_j, which is provided by the edge server. $P_D(cached)$ denotes the hit ratio, or rather the *future* hit ratio - the probability that the next request will be for one of the cached objects. To maximize this future hit ratio, the cache should contain models for the objects that are likely to be in the next request. PDF_j represents the probabilities of all possible next objects. Thus, $P_D(cached)$ should be an aggregate of probabilities selected from PDF_j. $P_D(cached)$ is expressed as, $P_D(cached) = \Sigma_k PDF_k$. This sum denotes that if the cache size is k, and the first k objects in PDF_j are kept in the cache, then the probability of a hit in the cache would the sum of the individual probabilities of those objects. By keeping

PDF_j ordered by the objects' probabilities, the most likely objects are always kept in the cache, increasing the value of $P_D(cached)$ and thus the hit ratio.

4.4.3 $L_{Net} + L_{Edge}$

L_{Net} represents the latency due the network and L_{Edge}, the lookup latency at the edge. On the device this can be measured as a whole, the total miss latency - the time it takes for recognition if a cache miss occurs on the device. Not only is this quantity easier to measure than the individual terms, it also captures other effects that impact latency. For example if requests are being queued at the edge then L_{Edge} might not capture the time spent in the queue.

Whenever a cache miss occurs, the measured response times from the edge cache is recorded, and the miss latency is estimated as a rolling average. This smooths out any spikes in the response times. Note, the request may miss in the edge cache and get forwarded to the cloud. This will add to the overall latency. This automatically gets incorporated in the measurements since it is measuring the total response time from the device's perspective.

4.5 Adaptive Feature Extraction

As introduced in Chapter 1, the recognition latency for a recognition pipeline is the total time consumed by both the feature extraction and classification (and optionally verification). Typically, for a given application, both of these phases are static. In Section 4.4.1 we show how the classification time can be controlled by modifying the size of the trained model on the device. Another approach to minimize recognition time is by minimizing the feature extraction time.

4.5.1 Controlling Feature Extraction Time

Feature extraction in itself is a complex process, and is controlled by a large number of parameters, which depend on the kind of feature extractor being used. The time it takes to extract features from an image depends on how "complex" these extractors are. Hence it is important to define a common measure for "complexity" which can then be used as a knob to affect the extraction time. In this thesis we will focus on local feature extractors such as SIFT [21], SURF [22] and ORB [23], since these extractors are commonly used for mobile object recognition applications such as augmented reality, object tracking, product recognition etc.

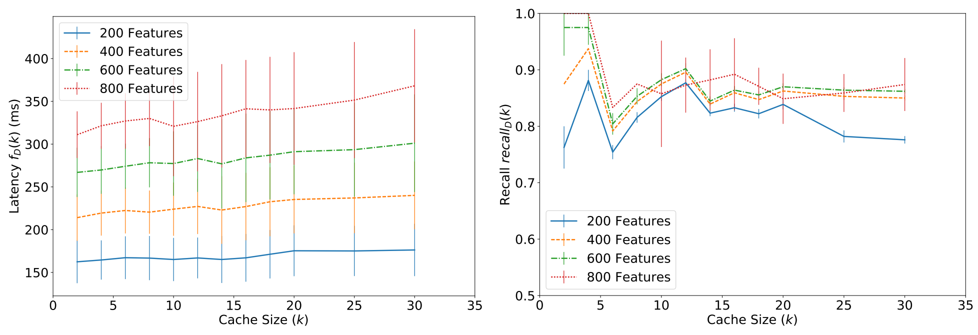
4.5.1.1 Local Feature Extraction

When using local features, multiple features are extracted from each request image, and then these features are used in the classification phase. The extraction of local image features is typically carried out in two major stages: (i) feature keypoint detection, followed by (ii) descriptor computation for each detected keypoint. With the advent of fast detectors such as FAST [56], and BRISK [57], ORB [23], the bulk of the time spent in extraction is in the descriptor computation. Each such detector has a set of parameters to configure how it extracts each feature from the image. A number of these parameters are specific to the detectors and control the efficiency and accuracy of feature detection and description. One parameter that is available across the detectors, which does not change the internal configuration of the parameters, is the *number of features* detected in each image. If the number of features detected is lowered, then fewer descriptor computations will need to be done.

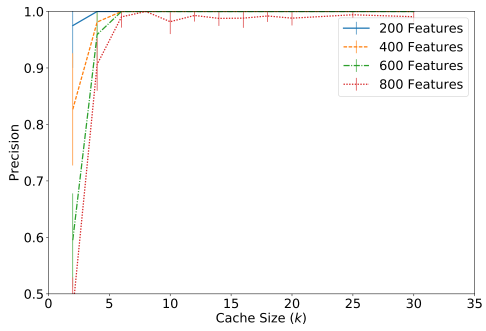
However, this also has a direct impact on the overall recognition accuracy. Recognition accuracy is dependent on the trained model being used for the classification and the number of extracted features. If input images need to be classified among a large number of object classes, then more extracted features from the input image will be required than when the number of object classes is lower, to maintain high accuracy. Since the cached model size will be controlled dynamically, we believe that Cachier will benefit by dynamically controlling the number of features extracted as well. To understand this tradeoff and decide the cache size and the number of features to extract at run time, we conduct a tradeoff analysis.

4.5.2 Offline Tradeoff analysis

The relationship between the recognition latency, recall, model size and number of extracted features is analyzed by conducting experiments in a similar manner as described in Section 4.4.1. That is, a given dataset is used, models for different model sizes are trained, and requests are classified using these trained models. In addition, to measure the effect of number of extracted features, for each request image, the request is repeated with a different number of extracted features. Recall, precision and recognition latency are measured for these experiments. Figure 4.4 shows the results of these experiments, using the ORB feature, LSH classifier and the Stanford dataset [53]. The analysis was conducted on mobile devices (ASUS Nexus 7) to ensure that the



(a) On-device recognition latency (b) On-device recall



(c) On-device precision

Figure 4.4: Latency and recall tradeoffs for different numbers of extracted features. Note that high precision is important to maintain accuracy.

measurements reflect what users would experience.

From the analysis, it is evident that lowering the number of features extracted significantly lowers the recognition latency. When extracting only 200 features, recognition time is very low. At the same time, the recall for that setting drops fast with increase in cache size (k). On the other hand, higher number of extracted features offers better recall for higher cache size, but is much slower. Note, here recall is an important metric not just for accuracy, but also for overall latency. If recall is low, it will lead to requests encountering recognition misses on the device's recognition cache and will be sent to edge server, increasing the overall latency. Thus these two parameters, number of extracted features and the cache size, k, need to be co-optimized such that overall latency is reduced.

4.5.3 Online Parameter Selection

The data collected from the offline tradeoff analysis needs to be utilized in making online decisions of how many features to extract, and what size should the cache be.

Number of features extracted	$f_D(k)$	$recall_D(k)$
200	$170.34 + 0.70k$	$0.81 - 0.00619k$
400	$221.24 + 1.13k$	$0.87 - 0.00324k$
600	$265.01 + 1.74k$	$0.88 - 0.00119k$
800	$310.99 + 2.05k$	$0.87 - 0.00146k$

Table 4.1: Estimated $f_D(k)$ and $recall_D(k)$ for different numbers of extracted features.

The online decisions are made based on the Equation (4.1). Hence the data from the tradeoff analysis needs to be incorporated here. To do so we turn back to $f_D(k)$ and $recall_D(k)$. Note that the measurements we made in the tradeoff analysis, recognition latency and recall for different k, correspond to these two terms in Equation (4.1). So we can use the measurements made to estimate $f_D(k)$ and $recall_D(k)$ using regression analysis. To incorporate the effect of changing the number of features extracted, multiple pairs of polynomials were estimated, for each possible number of features, as shown in Table 4.1.

At run time, multiple versions of Equation (4.1) are generated using each pair of polynomials. These are minimized in parallel, individually, and a minimum is then found across all the versions. The number of features for that version is used for future requests, and the k that minimized that version is used as the cache size.

4.6 Minimizing Expected Latency

Figure 1.2 depicts Cachier's system architecture, split across the devices and the edge server. The different components across the system implement the aforementioned techniques to estimate Equation (4.1), find the value for k that minimizes it, and the prefetch the corresponding parts of the model onto the device.

On the left is the recognition pipeline on the device. This is similar to the pipeline described in Section 1.1. The application submits images (or frames from a live video stream) to the pipeline, and expects to see the recognition result as the output. Typically, the inner modules of such a recognition pipeline are static. To minimize recognition latency, Cachier dynamically changes the parameters in this pipeline, specifically in the feature extraction and classification phases. The Optimizer module leads this decision making process. It uses information from different sources, namely offline analysis, edge server's Markov estimator, and the Profiler modules, to decide the number of features to extract in the feature extraction phase, and which objects to include in the

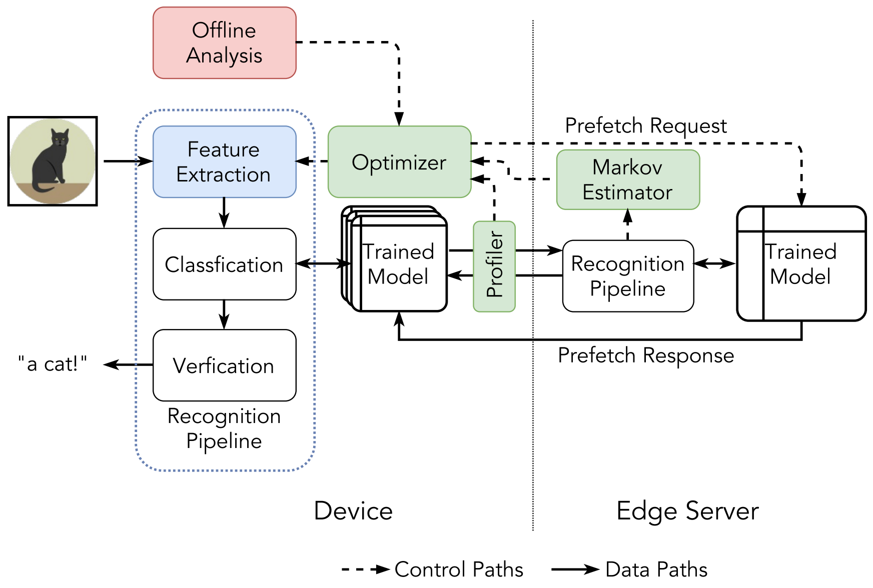


Figure 4.5: Cachier's Architecture.

cached trained model on the device.

When an image arrives into the recognition pipeline on the device, it proceeds as it normally would - features are extracted using the current parameters, the extracted features are classified using the current trained model, and this is followed by geometric verification. Cachier's features kick in differently for two scenarios - cache miss and cache hit.

Cache miss. On a cache miss, either compulsory or accuracy-based, the image is sent to the edge server to be recognized. On the edge server, once the image is recognized, the Markov estimator updates its estimates (Section 4.3.1) and the response of the recognition is sent back to the device, along with future object probabilities. The Profiler on the device measures the latency of the response and stores it. The Optimizer uses the probabilities in the response, measurements from the Profiler and data from the offline analysis to minimize the expected latency (Section 4.4). It uses gradient descent to find the cache size and number of extracted features that minimize the future expected latency. Depending on the results of the optimization, the Optimizer reconfigures the feature extraction phase (Section 4.5) and prefetches required objects into the device's cached model.

Cache hit. On a hit in the cached model, the Optimizer requests for the probabilities of the next possible objects from the Markov estimator in the edge server. Then, similarly as before, using information from different modules, the Optimizer runs the minimization and reconfigures the feature extraction phase and cached model if required.

In this manner, the recognition cache on the device is updated after each request. Note, Figure 4.5 depicts multiple stacked trained models. This is a system optimization to and is discussed next.

4.7 Optimized model loading

Since Cachier prefetches and updates the cached model based on what the current object is, when the current object changes, the cached model will also change. Even though the model is small, it can take hundreds of milliseconds to fetch and load the model and begin using it on the device. This can delay recognition on the device. Coupled with this, it is possible that the user quickly switches between a small set of objects, e.g. comparing prices of cereal boxes or comparing paintings of a painter. When this happens, the cache will keep switching between models based on what the user is currently seeing. This can potentially cause further delays in recognition. To overcome these challenges, we employ two techniques.

4.7.1 Immutable data structure

Cachier uses an immutable data structure to store the cached model. When the current object changes and the model needs to be updated, it allocates memory and loads the new model. This is done on a separate thread. If any requests arrive in this window of few hundred milliseconds, they are looked up using the previous model. Given the small time window, these requests will most likely contain the new current object. Since the cached models are trained to recognize future requests, the previous model should be able to recognize the new current object in the requests. Here, we rely on the assumption that users will dwell on any object for at least a few hundred milliseconds, and the model can be updated in the background. Once the model is loaded, it is atomically made the current model and all requests are then classified using the new model.

4.7.2 Caching models

When a user is in the "comparison" mode as described above, it can stress the resource constrained mobile device by constantly switching models. Cachier would keep allocating new chunks of memory to load the model. To avoid this, Cachier employs a software LRU cache for the models.

Whenever a model is loaded into memory, it is inserted in the LRU cache, with the current item's ID as the key. Whenever the current item changes, Cachier checks if the LRU cache contains the new item's ID and the corresponding model. If it does, this model can be used directly, without the need to allocate new memory. Cachier restricts the size of this LRU cache to 4.

Chapter 5

Cooperation between Devices

In the previous chapters we have discussed methods to reduce recognition latency by using edge servers on their own or in collaboration with devices. When used in collaboration with devices, edge servers have provided the analytics that helped reduce the computational burden on the devices. Thus edge servers have played a key role in our approaches. We have *assumed* that such servers will always be easily available near edge devices, since they have not actually been deployed yet. However, there can be multiple scenarios in which they are unavailable or unusable. For example, edge servers are being pushed for deployment in mobile cellular base stations rather than Wi-Fi access points. In fact, it is being argued that it is better to deploy edge servers at aggregation points, where networks from multiple base stations converge [58]. This will have multiple ramifications. First, this will increase the latency between edge servers and devices, which will make the devices do more computation locally. Secondly, such deployment scenarios will put large areas and number of users under the same edge server. This will decrease the spatiotemporal locality across the users, forward more requests to the cloud and increase response times. Moreover, this will also burden the cellular wireless spectrum being utilized between the edge servers and the devices.

Even if edge servers are close to devices, there will still be opportunities to leverage finer spatiotemporal locality. For example if one server is connected to the network controller of a building, it will be respond to requests from everyone in the building and leverage the locality shred by all the users, irrespective of where they are within the building. However users in the same room or users on the same floor of the building will certainly share more context, than users

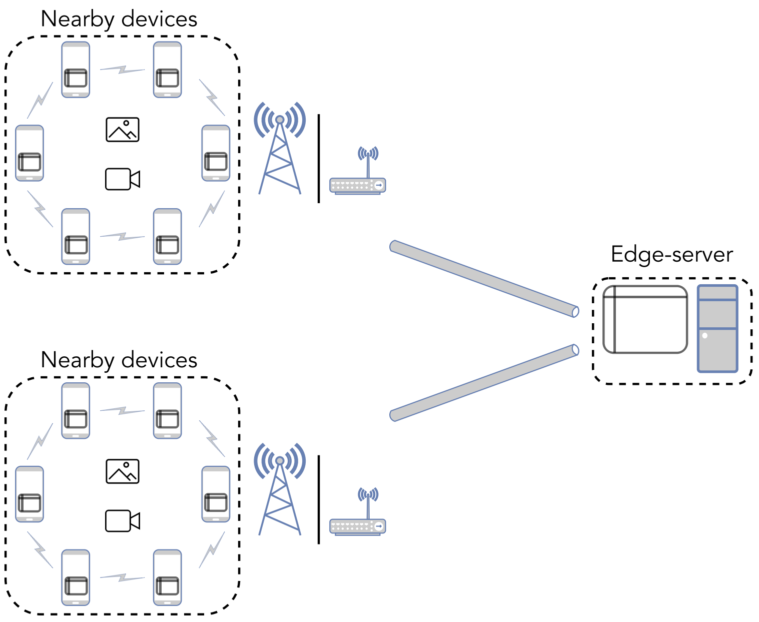


Figure 5.1: Groups of nearby devices see similar objects and share spatiotemporal context.

in different parts of the building, but the edge server might not be able to leverage locality at such granularity.

We believe that collaboration and sharing information directly among neighboring devices can help leverage locality at granular levels. We propose a distributed key-value store that is supported by the devices themselves, for sharing local, on-device information. Mobile devices have communication capabilities which allow them to talk to other nearby devices without going over the Internet, such as wireless local area networks (WLANs) and Bluetooth. Since these wireless capabilities use a different wireless spectrum, using them can lower the stress on the cellular wireless spectrum. Our approach leverages this and handles the communication and coordination between nearby mobile devices to provide an easy-to-use key-value store abstraction for sharing information. A key-value store will make it easy to find and retrieve information, without the need to broadcast to all devices.

5.1 Leveraging Granular Locality

Users and devices that are immediately near each other, share a high degree of spatiotemporal locality. These groups of edge devices see the same objects, under the same conditions, in a short time interval. Such visual overlap can be leveraged to enable mechanisms that contribute towards better, faster recognition applications. These mechanisms can not only take over an absent or distant edge server's responsibilities, but can themselves be novel as well. Here we list some such mechanisms that can be enabled if nearby devices cooperate with each other.

5.1.1 Cooperative Prefetching

If the edge server is situated in cellular network basestations, then the latency between it and the devices will be significant [58]. Such basestations also serve a much larger user base than a Wi-Fi access point, because of the area they are meant to cover. This combination of higher latency and more devices can lead to significant delays when multiple devices request data from the edge server simultaneously. This can directly effect the prefetching technique presented in Chapter 4. When each device connected to the edge server retrieves models from the edge server to process future requests, it can lead to congestion and delays. Instead, it is possible to leverage locality to minimize such downloads from the edge server.

As mentioned earlier, neighboring devices and users will likely see the same objects within short time windows. This implies that the devices will require models for the same objects. If these devices could cooperate and communicate, it would be possible to share such models directly with each other. This will minimize downloads from the edge server.

5.1.1.1 Model Adaptation

As the devices extract features from and classify new images, they can start learning from these newly classified features as well [59, 28]. This way they can update the model with the new features, which will make the model more sensitive to the current lighting and environment conditions, which are likely different from the data that the model was trained on. Then by sharing this adapted model, this model can benefit all nearby devices.

5.1.2 Cooperative Analytics

Apart from sharing models directly, cooperation among devices can also enable granular analytics which can potentially lower recognition latency on the devices.

5.1.2.1 Popularity Estimation

In previous chapters we discussed how the edge server can estimate request distributions to assess which objects are popular in the area that they serve. If the edge server serves a large area, or if there is no edge server, this estimation can be carried out by the devices themselves. If neighboring devices can cooperate, they can aggregate individual counts of objects that each device has seen, and rank the objects according to popularity.

This can be taken a step further by estimating a Markov model in a similar manner. The estimation process is similar to the one described in the previous chapter, but now the model is more granular. Here again, by cooperation, devices can keep counts of object pairs to estimate the Markov model.

5.2 Distributed Key-Value Store

An infrastructure or middleware system is required that can enable all of the above potential efficiency techniques. This system should provide mechanisms to search for data across the nearby devices and to aggregate information when required. The system should handle the device-to-device communication and cooperation required to provide these mechanisms. We believe that a distributed key-value store, where each device is a node in the store, will be apt for such a system. It will make data searchable through keys, and aggregators can easily be implemented as values in such a system. A device can publish relevant data that it has locally, and it will be available to other nearby devices to fetch and use. For example, to share prefetched information, a device can *put* the model for the object with the object ID as the key. Then when a device needs to prefetch an object's model, it can use the object's ID as the key to *get* it. Popularity and Markov model estimation can be carried out similarly with object ID being keys, and aggregators being the corresponding values.

5.3 System Design

This key-value store should have the following high-level properties:

Efficiency. The system itself should have low overheads with respect to bandwidth.

Responsiveness. Since the system is suppose to lower response times, it search or information requests should have low end-to-end latency.

The basic goal of the system is to provide a distributed key-value store to enable discovery, sharing and retrieval of information across nearby devices. This is achieved through the use of network-based discovery to find nearby devices and consistent hashing to guide the `put` and `get` requests made on the key-store to the correct nearby device.

Guiding principles for the overall design are decentralization, to avoid any one device becoming a central hub or coordinator, and autonomy, since there can be no system administrators for such a system and the it needs to manage itself.

Defining "nearby". We define "nearby" devices as those that are one hop away on the same network such as devices connected to the same wireless access points (WAP) or devices using Wi-Fi Direct groups. Given the prevalence of such access points and development of technologies such as Wi-Fi Direct, we believe this is a pragmatic assumption, and provides a number of features that can be leveraged.

5.3.1 Background: Distributed Hash Tables

Distributed hash tables (DHTs) [60, 61] are a class of structured peer-to-peer systems. They use key-based routing and consistent hashing to provide a lookup service similar to a hash table, but where the (key, value) pairs are distributed across participating nodes. They were designed to function on peers (desktops) distributed across the Internet, for cooperative Web caching, distributed file systems, domain name services, instant messaging, multicast, and also peer-to-peer file sharing. On the surface, Cachier and DHTs look similar and in fact, Cachier draws inspiration from DHTs. Both provide a key-value based lookup service, route based on keys, use some form of hashing, and distribute the workload across all participating nodes. The key design choices were made for DHTs to work across the wired Internet. However, the goal for Cachier is to work across a *local, wireless* network, and this is the key difference.

5.3.2 Decentralized Discovery and Presence

For a device to join a group of nearby devices, it needs to announce its presence, and discover other devices. Typically this is done through a central registry or bootstrap server. We propose to do it without the need of one. We leverage the fact that these devices are connected to the same WAP. This allows each device to receive broadcasts from every other device connected to the same access point. Typically broadcasts on wired networks are costly, requiring n^2 messages to traverse the network when n devices are connected to it. But in a wireless network, transmissions are natively broadcast in nature. Wireless access points have the capability to broadcast a single packet to all connected devices. By leveraging this, each device is only required to transmit one message, thus a total of n messages are required for n devices. By transmitting this message periodically, this discovery mechanism is also utilized for presence detection.

When a device joins a WAP, it starts broadcasting these *beacons* to alert other devices about its presence. It sends UDP beacons to the broadcast address once every five seconds. At the same time, the device also listens for beacons from other nearby devices. The beacon contains the network endpoint that the sender device is listening on. Thus on receiving a beacon, a TCP connection is made with the beacon's source. Since these beacons are periodic, they are used for detecting device departures as well. If a device does not receive a beacon (or any other data) from another device that was discovered earlier, for T seconds, where T is configurable, the former device tries to ping the latter over the formed TCP connection. If this is unsuccessful, the device is considered to have left the vicinity. This is done by all the devices in the group. Therefore at steady state each device eventually has a list of all other devices in the group, i.e. connected to the same access point.

It might seem like this discovery scheme uses a large share of bandwidth. However, by keeping the beacons small (13 bytes on average) and using the broadcast mechanism, the bandwidth consumed is approximately 255 bytes/s per device, for a group of 20 devices and broadcast period of five seconds, which amounts to less than 0.01% of a nominal 54MB/s Wi-Fi network. By using this discovery mechanism and tracking devices' presence, each device can keep an up to date list of available devices and manage connectivity to those devices. This preserves symmetry and avoids having a centralized registry for storing membership and device presence information.

5.3.3 Hash-based Routing

As this is a distributed key-value store, the storage for the key-value pairs is provided by all the neighboring devices. To keep the workload equal amongst participating devices, the key-value store needs to be distributed and decentralized. A hashing scheme is used to partition and distribute the keys of the key-value store. This ensures each device is responsible for a unique subset of the keys and no single device is overloaded.

5.3.3.1 Rendezvous Hashing

The hashing scheme used by Cachier is *rendezvous hashing*, also known as highest random weight hashing [62]. This algorithm allows the devices to achieve distributed agreement on where to place objects, independently, i.e. without the need to run a consensus protocol. Given a key and a list of known nodes, this algorithm consistently orders the list of nodes. It uses hashing function, $h()$, that consistently generates weights for each node, using the key and the node's name or ID. It then orders the the list of nodes in descending order of the generated weights. Then the first node can be assigned the key that was used for this ordering. This can even be extended so that if replication is required, the top-k nodes can chosen to store the key. This algorithm can be run independently on each node, and since $h()$ is the same on all nodes, the same order of nodes are generated for the same key. In [62], this scheme was used by clients trying to access resources over the Internet. If all the clients had the same list of servers, they would all always fetch a resource at a given URL from the same server.

The rendezvous hashing scheme ensures that the system is decentralized yet coordinated, and the workload is distributed. As there is no single “master" device, lookups and stores are spread across all devices and no node becomes a bottleneck. The use of rendezvous hashing helps minimize communication overhead for the system since it obviates the need for a consensus protocol. By using rendezvous hashing, the system also ensures low end-to-end latency. Unlike DHTs such as [60, 61], all lookups in this scheme are resolved in one hop, hence key retrieval is fast. This is an appropriate choice for the number of devices in nearby groups. Since WAPs allow for 40-50 active connections at a time, there is no need of using multiple hops, as used in [60, 61], to reach the destination.

The system uses the set of available nearby devices provided by the discovery module, and consistently maps a key to a device. Given that the discovery module on each device discovers every other device in the group, this hashing scheme will always map a key to the same device.

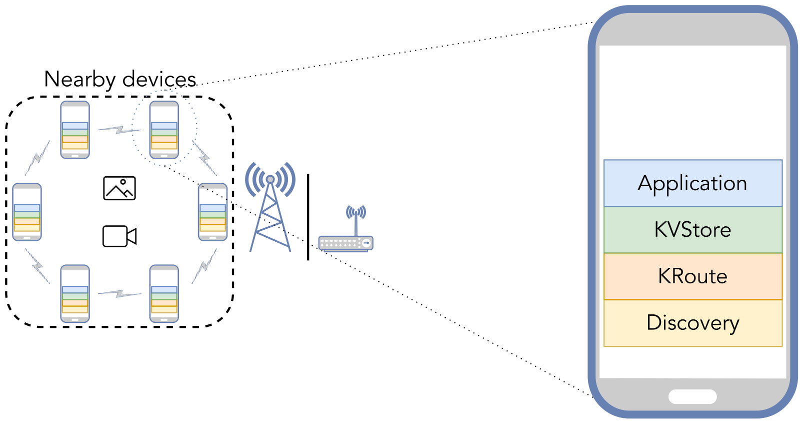


Figure 5.2: Nearby devices around a wireless access point, where each device runs the system shown on the right.

5.4 System Architecture

As can be seen in Figure 5.2, the system is divided into three main components: Discovery and Communication, KRoute, and KVStore; the application, i.e. the mechanisms that share data or aggregate information, on top interacts only with KVStore. The modular structure allows for addition of more features in the future, and also allows direct access to lower layers if need be. An instance of the system and the application runs on each participating device in the group.

5.4.1 Discovery and Communications

This module implements the discovery and presence mechanisms. It also forms the TCP connections with other devices and provides an asynchronous communication abstraction to the above layer. It discovers new devices and keeps an up-to-date list of available nearby devices, which it provides to the above layer.

5.4.2 KRoute: Partitioning and routing

KRoute implements the hashing scheme. It presents itself to other components as a high-level communication layer, where instead of requiring endpoints to communicate with, it requires keys. It uses the hashing mechanism to map keys to devices. Hence, if two different devices provide the same key to their respective KRoute modules, they end up communicating with the same device. This is essential for efficient coordination. It obviates the need for any dynamic agreement protocol when multiple devices want to talk to the same device. Neither does it need multiple hops to find the correct device. This is important because multiple hops can cause congestion at the WAP.

By keeping the hashing mechanism at the communication layer, it can be used for more than just distributed storage. In effect it allows an SDN (software defined network) abstraction. If applications want to communicate with a known service on the network, it would just need to use the service name or ID to communicate with it. Moreover, the system allows for explicit replication. If the application needs to send a copy of the message to multiple entities, it needs to only specify how many replicas. Since the hashing mechanism provides consistently ordered list of devices, the same message can be replicated to top-k devices, for k replicas.

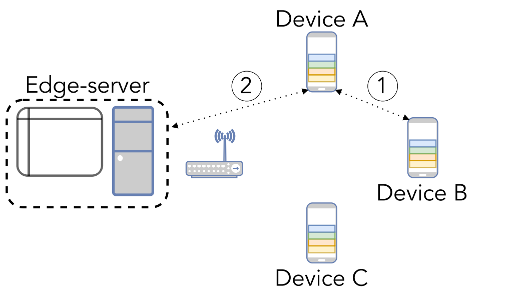
5.4.3 KVStore: Key-Value Abstraction

Armed with the key-based communication provided by KRoute, this module builds a thin remote procedure call (RPC) layer on top of it. It also contains the device-local hash table for storage of keys and values. The RPC layer implements the two procedures required for a key-value store, namely *put(key, value)* and *get(key)*. When the application stores a key-value pair, KVStore creates and serializes a *put* RPC object and sends it using KRoute by providing the key from the key-value pair. On receiving a remote *put* call, KVStore extracts the key-value pair from the serialized RPC and stores it in its local hash table. When the application requests a key, KVStore provides the *get* RPC object and the requested key to KRoute, which sends it to the correct remote device. On the remote device, when KVStore receives the RPC, it extracts the key, checks its local hash table and replies with the associated value. By providing a simple key-value interface, KVStore makes Cachier easy to use for applications.

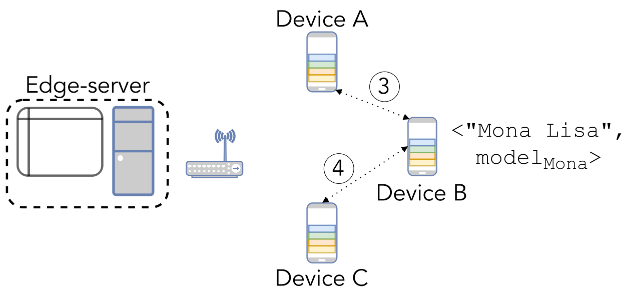
There are two ways in which KVStore can be used: (1) directly store items in the key-value store, or (2) use as a redirection service.

5.4.3.1 Direct storage

For directly storing items, attributes of the information such as object names or IDs can use be used as keys, and the value is the information itself, such as the model or features or counts.



(a) The requesting device first checks if the assigned device has the model. If it doesn't, it fetches it from the edge server.



(b) After fetching it from the edge server, it puts the model in the chosen device. Future requests can fetch the model from this device.

Figure 5.3: Using the system to share models prefetched from the edge server.

5.4.3.2 Redirection service

To use as a redirection service, the keys are still the same, i.e. some attribute of the data, but the value will be the URI (Universal Resource Identifier) of the information. The URI is provided by the system, and hence can be resolved within that group. When an application issues a *get*, it gets the URI as the response. The application can then use the system to resolve the URI and fetch the data it points to. This is really useful when the information is large in size, e.g. a long list of extracted features, as we will see later on that this can reduce bandwidth consumption.

In both cases, if other users also store values with the same key, these values can be configured to be appended into lists. When fetching, the application would need to provide an attribute for the content it is looking for and would receive the consolidated list.

5.4.4 Using the system

As we mentioned, the system can be used for multiple different mechanisms that can increase efficiency, or make the system less dependent on the edge server. We highlight one of the mechanisms here and show how the system can be used.

5.4.4.1 Enabling Cooperative Prefetching

We discussed in Section 5.1.1 that it will be more efficient if a device could share the information it has prefetched with other nearby devices. This would decrease the amount of data that needs to be downloaded from the edge server. We describe how this can be achieved by the key-value (k-v) store. Figure 5.3 shows the flow of information.

Suppose Cachier on device A wants to prefetch the model for the object "Mona Lisa". It will first check the k-v store by using the `get("Mona Lisa")` RPC on KVStore. KRoute will take the key and the list of available devices and hash them. Suppose in the resulting list device B is first. KRoute will send the RPC to device B. Since device B does not have this key yet, it will return a null. On getting this response, device A will fetch the model from the edge server. Note, there is no need to check any other device because only device B could have had that key. Once device A has the model, it will insert it into its local recognition cache , and then issue a `put("Mona Lisa",` $model_{Mona}$`)`. This will use the RPC and contact device B. Device B will store the model in its local k-v store. Now suppose device C wants to prefetch the model for the object “Mona Lisa". It will first check the k-v store by using the `get("Mona Lisa")` RPC on KVStore. This will get routed to device B again, where the model will be found. Then in the response of the RPC, device C will receive the model, which it will insert into its recognition cache.

Potentially, only one device per group may need to download a model for a given object from the edge server. The other devices can get a copy of the model from this on device, device B in this case. If instead of using direct storage, the system uses redirection, this load can be balanced. Then device A would have stored the URI on device B and device C would receive this URI, which it would use to fetch the model from device A. On successfully fetching the model, it would enter the URI for its copy of the model into the k-v store, which would be appended into the list on device B. Thus as more devices store the model locally, they too can become sources and thus the load can spread.

Chapter 6

Implementation and Evaluation

6.1 Implementation

Cachier is implemented as a recognition service with a software development kit (SDK), similar to [15, 46, 16]. The application developer provides the images of objects which the application should recognize. This is used as the training set to build the model in the cloud. The SDK has a library that is installed on the mobile device along with the application. To recognize objects in an image or video frame, the application calls the `recognize(image, callback)` function, providing the image to be recognized and the callback. The callback is used to notify the application about the result of the recognition.

The interactions between the mobile device, edge server and the cloud are over remote procedure calls (RPC). We implement the RPC to allow extensive logging and measurements, and it uses ZeroMQ [63] internally for asynchronous communication. The interface definition for the RPCs and the serialization is implemented through Google Protocol Buffers [64]. All of these details are abstracted away from the application through the use of the SDK.

Cachier and the entire recognition service is implemented in Java. The mobile component is implemented for the Android operating system, while the servers, both edge and cloud, are implemented for the Linux operating system. All the image manipulation and recognition tasks across the system use implementations of algorithms from the open-source computer vision library, OpenCV [65].

In this implementation, the two key algorithms used for the recognition pipeline are ORB [23] for feature extraction and multi-probe LSH [52] for feature classification. Compared to others, ORB allows for fast extraction of binary features, while LSH allows scalable feature classification. The choice of the feature classification algorithms is important since that dictates how the cached model is trained. In the case of LSH, to train the classifier to be able to recognize an object, the features of that object are inserted into the hashtables created by LSH. Thus to recognize objects on the device, Cachier creates an LSH classifier on the device, fetches features of the objects being prefetched from the edge server, and inserts them into the device's LSH classifier.

6.2 Evaluating the Effect of Scale

The goal of Cachier is to allow large scale use of recognition applications, while at the same minimize the response time for each user. Thus the goal of the evaluation is to measure the effects of different kinds of loads on Cachier, explore the operational space and to evaluate how each proposed technique helps alleviate the load, enable higher scalability and reduce response time.

There are various different variables that can impact the performance and efficiency of a mobile recognition application, such as the nature of the application itself, or application-independent factors such as network availability. This leads to a wide variety of operational points at which recognition applications might function. In this evaluation we will run multiple experiments that explore this operational space to analyze the implications on recognition applications and on the proposed techniques and present the trade-offs that are made to lower end-to-end latency.

6.2.1 Variables

At the very outset of the thesis we had described that the biggest challenge that we wanted to address was how will systems deliver real-time recognition in such a device-dense world. Thus, though there are multiple variables that can potentially impact applications, we choose a set of variables that demonstrate the effect of scale on systems that support recognition applications.

6.2.1.1 Number of users

This controls the number of edge devices or users simultaneously using the application in a location served by an edge server. Given the rapid increase in the number of people carrying mobile devices, it is important to know how/if recognition systems will scale to accommodate them.

6.2.1.2 Object density

This controls the number of recognizable objects in the area served by an edge server. Edge servers can be configured (manually or automatically) to only recognize objects in their area, thus making this an important variable to analyze an edge server's performance. This also affects compute time and accuracy.

6.2.1.3 Total number of classes

This controls the size of the set of all objects the application is supposed to recognize. This depends on the type of application - e.g. a smart assistant may need to recognize a large number of objects while product-specific recognition applications may recognize a much smaller set. This will affect compute time and accuracy.

6.2.1.4 Network availability

This controls the network conditions between the edge server and the cloud. The motivation for edge computing is based on the premise that this part of the network will become the bottleneck for recognition applications and hence it is vital to see how network conditions impact an application's performance.

6.2.2 Metrics

To assess the impact of the variety of loads, a number of metrics need to be measured in each experiment.

6.2.2.1 Latency

End-to-end latency - the time between capture of the image and reception of response on the device. Along with measuring end-to-end latency, break-downs of time spent in computation and communication individually is also be measured.

6.2.2.2 Accuracy

Accuracy of the recognition system is measured through its recall and precision. Since the recognition cache also runs recognition algorithms, its effect on the accuracy of the overall system needs to be evaluated.

6.2.2.3 Infrastructure resource utilization

There are two types of infrastructure resources which are utilized in recognition systems - compute resources and the network, measured through bandwidth consumption. Compute resources are used in both, the edge server and cloud. Since the goal of the proposed techniques is to enable support for more devices, we need to analyze whether the techniques add or remove stress from the infrastructure.

6.2.3 Configurations

By comparing the metrics measured for all the experiments for each system configuration, we will be able to shed light on the trade-offs the contributions make to achieve low latency. For example, we expect that there for few devices, on-device local computation might not be necessary. The sections below describe all the different system configurations for which the experiments will be performed. Italicized configurations are our novel contributions.

6.2.3.1 Only Cloud

The typical image recognition approach for mobile applications. All images captured by each mobile device are uploaded to the cloud for processing.

6.2.3.2 *Load-balanced Edge Cache*

An edge server is placed in the system above, in-between the devices and the cloud, and uses a recognition cache. The recognition cache employs all the techniques described in Chapter 3 and uses the load-balancer to support multiple devices.

6.2.3.3 *Bounded Edge Cache*

The cache in the edge server is similar to the above but for multi-user support, it dynamically bounds the allowed compute time and thus the cache size, as described in Section 3.5.1.

6.2.3.4 *Prefetching and On-device Computation*

On-device computation is added to the above system and prediction is used to decide which objects to prefetch and compute on the device.

6.2.3.5 *Adaptive Feature Extraction*

On-device recognition is modulated to not only change the feature classification phase, but also the feature extraction phase.

6.2.4 Application

An application is required to drive the above analysis. A desirable application will be one in which all the variables can actually change in the real world and requires low-latency responses. There are a number of applications which exhibit such properties. For example,

- A product search application, like [45, 4], that can be used in grocery stores, which uses the phone's camera to locate items on shelves and retrieve nutritional and allergen information.
- A smart virtual assistant for smartphones, like [66, 67], in which the users can point the camera at objects around them and get contextual information. For example trivia about buildings and monuments, showtimes in a movie theatre, or names of flowers.
- An indoor navigation application, like [3], which can guide first-time or vision-impaired visitors, or even drones inside buildings, where GPS does not work.

There are some applications that do employ recognition, but due to other constraints, are not amenable for distributed operation. For example, recognition of vital features such as lane markers, pedestrians, other cars etc. in autonomous vehicles have hard realtime constraints and hence are not suitable for offloading and will be executed on the vehicle itself.

A good representative application in this application space is an art recognition application that recognizes artifacts and paintings in museums and galleries and provides visitors with new content or information, overlaid on top of the camera's view, like [44, 68]. Such augmented reality applications are known to have low-latency requirements [17]. Mapping the aforementioned variables to such an application:

- The network conditions would be the prevalent network conditions between on-premise edge servers and the cloud.
- The total dataset would be all the artifacts in a museum.
- The edge servers would serve their own respective sections of the museum and thus object density would be the number items in those sections.
- The number of users would be the number of visitors in a given section of the museum.

To drive the experimental analysis we have developed such an art recognition application. This application runs on the mobile devices and makes recognition requests during the experiments to recognize artifacts in images. We first describe the dataset for such an application and then the setup for the experiments.

6.2.5 Dataset

To evaluate a recognition system, using actual traces of users using such an art recognition application would provide the best results. However, such traces do not exist in the public domain. Instead we have built a dataset using museum visitors' mobility traces, overlaid with painting recognition requests, to yield recognition traces that emulate a trace from an art recognition application.

From [1], we take real-world mobility traces that capture the mobility of 180 visitors walking around a room of a museum that contains 44 exhibits, as shown in Figure 6.1. Each trace in this dataset tracks a unique visitor as they move around in the room. Each entry in the trace contains three items: the time at which the visitor starts looking at an exhibit, the duration for which they look at the it, and the ID of the exhibit being looked at. To create a recognition trace using a visitor trace, the visitor trace needs to be augmented with image requests. We use a public image dataset containing images of exhibits from a museum along with other objects [53] to create the image requests. Some sample images from the dataset are shown in Figure 6.2. This dataset provides both training and test images. To create a recognition trace, first, the exhibit IDs from the

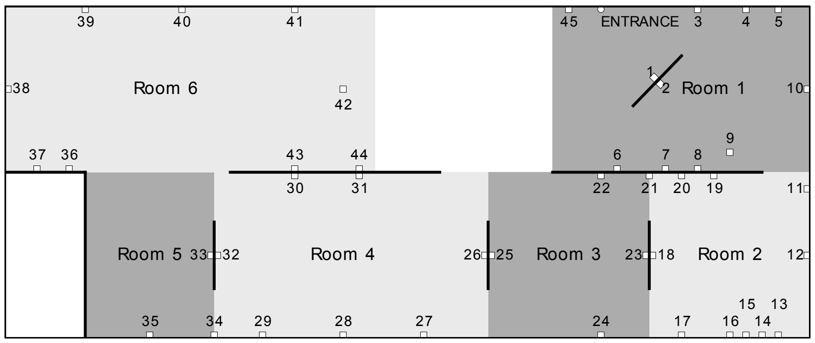


Figure 6.1: Floor-map of the section in the museum from which visitor traces were taken. Each number is an exhibit ID, and its position represents the location of the exhibit. Image taken from [1].

Figure 6.2: A selection of images from the dataset which are used as request images in traces.

museum traces are mapped to objects in the image dataset. Then, in each trace, for each second that a visitor is looking at an exhibit, a training image for the mapped exhibit's ID is sampled from the image dataset and used as an image request. Such traces mimic what the visitors might be seeing as they walk around the museum room. Moreover, given that it is taken from a real museum, the intra-object relationships of how actual visitors move from one object to another is also captured by the traces.

We use one frame per second instead of a typical framerate such as 30 fps, because there are many approaches that have been proposed to filter frames at the source. For example [28] proposes to drop frames based on readings from inertial sensors, [69] proposes to drop frames if they are blurry, and [29] proposes to make recognition requests only if the frame is significantly different from the previous frame. These techniques lower the number of frames that actually need to be recognized by the recognition pipeline.

6.2.5.1 Controlling the number of exhibits

The set of traces as is has the visitors looking at 44 objects in one section of the museum. In our experiments, we need to be able to control the object density, i.e. the number of exhibits that fall in the area covered by an edge server, to be able to see how this impacts the performance in the edge server. Hence the number of objects cannot be fixed at 44 and needs to be controllable. At the same time, we do not want to change the flow of any trace internally since that reflects the spatiotemporal locality. Instead, we use this section, shown in Figure 6.1, as a repeatable, indivisible unit. To increase the number of objects, the number of sections are increased, where the layout is the same. The exhibits, however, change - each new section is mapped to a new set of objects from the dataset. The visitors are equally divided across the sections. This allows us to keep the flow of the visitor in the trace, but still increase the total number of sections served by an edge server, and thus increase the number of objects served by the edge server.

6.2.5.2 Learning the Markov model

For the on-device computation configuration, models are prefetched from the edge server on the device. For prefetching to be successful in this configuration, the edge server needs to estimate a Markov model that can predict which objects will be queried next. We use the traces generated above, to have the system estimate a Markov model, and then we test the model. More than an evaluation of the system's estimation techniques, it is a verification of whether the traces exhibit the Markov property, and what order Markov model is best suited for these traces. We use 80% of the traces to train the model, and 20% to test. To assign an accuracy score, for each query in the test trace, Cachier's estimated model is used to predict the top 5 possible exhibits that might be next. The output is then checked to see if the actual next exhibit in the trace is present in the prediction. We run this experiment for different orders of the Markov model. The result, Figure 6.3, shows that the first order Markov model is indeed the best choice. The other models tend to overfit to training data and hence have low accuracy on test data. The accuracy for the first order model actually grows fast, for example in a test predict top 8, the accuracy is over 80%.

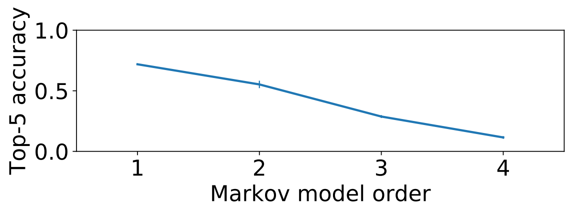


Figure 6.3: Accuracy of different Markov models.

	Mobile Device	Edge server	Cloud
Type	Nexus 7 2013	PC	Server
CPU Cores	4	8	24
Memory	2GB	8 GB	32 GB
CPU Speed	1.5 GHz	2.20 GHz	2.00 GHz

Table 6.1: Specifications of the mobile device, edge server and cloud server.

6.2.6 Setup

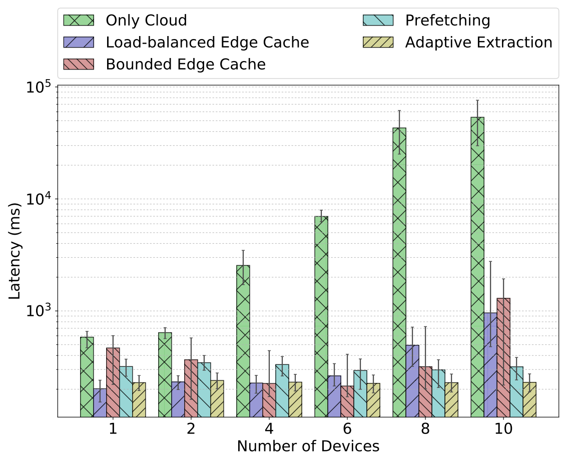
The characteristics of the devices, edge server, and the cloud server are given in Table 6.1. There are two different networks used in the system. There are a total of 10 devices that connect to the edge server over WiFi (802.11n), through a router capable of supporting 900 Mbps. The edge server connects to the cloud over gigabit ethernet. This network is controlled to emulate desired network conditions using `netem` [70].

6.2.7 Procedure

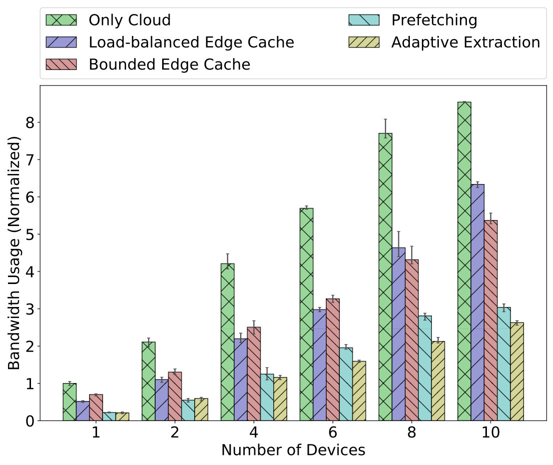
First, the cloud is started with a model trained using all objects for that experiment setting. Then the edge server is started, if present in the experimental configuration, and its cache is warmed according to the experiment setting. For the prefetching configuration, the trained Markov model is loaded on to the edge server. Each recognition trace is for a unique visitor and is assigned a unique label. For each experiment, each device is assigned a visitor. The matching trace is downloaded to the device. As mentioned above, the trace contains the ID of the training image to request, and the time at which to request it. The request images are stored on the devices as well. Once the experiment starts, the application on the device makes a request every second, according to the trace, using Cachier's `recognize(image, callback)` function. This is done asynchronously to ensure that the requests go out on time. The application then waits until it receives all responses. Each experiment is run four times, each time using a different set of traces for the

Number of Visitors	Number of Sections	Total number of Exhibits	Network Conditions
5	3	500	30ms RTT, 15 Mbps

Table 6.2: Baseline values for the variables.



(a) Latency or response time.



(b) Bandwidth utilization.

Figure 6.4: The effect of varying number of visitors on the different system configurations.

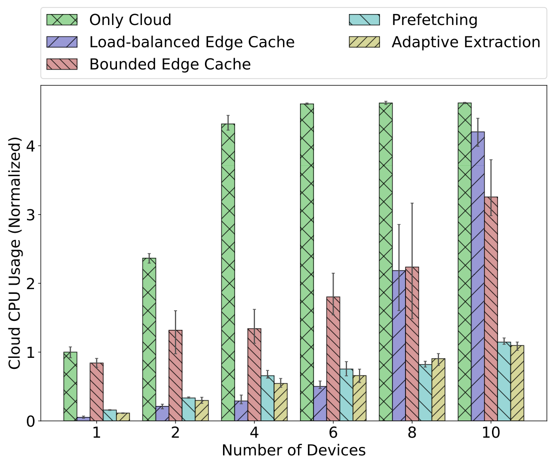
devices. In each run, all the configurations are run using the same set of traces.

Since there are multiple variables in the set of experiments, we define a baseline, and then vary one in each experiment while the others are held constant. Table 6.2 defines the baseline.

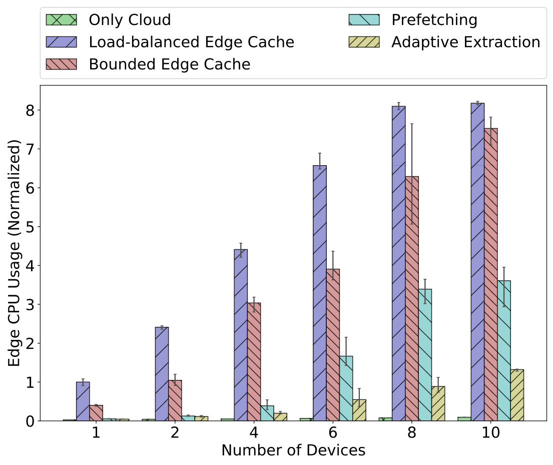
6.3 Impact of User Density

This experiment evaluates the effect of request load, that is how the number of users simultaneously using the application affects the application latency or response time. Thus in this experiment, we control the number of users (devices) that simultaneously send requests. All the other variables are kept at the baseline values.

We see that as the number of visitors increase, the response times on most system configurations increase, but the only-cloud system increases significantly faster than the rest. This happens for two reasons. First, as the number of visitors increases, the amount of data being offloaded increases, and all of it goes to the cloud. This causes congestion in the network, which leads to delays. Second, even though the cloud has formidable compute resources, it still is processing a model for all 500 objects in the dataset. This means that compute times for each request is high,



(a) Cloud CPU Usage.



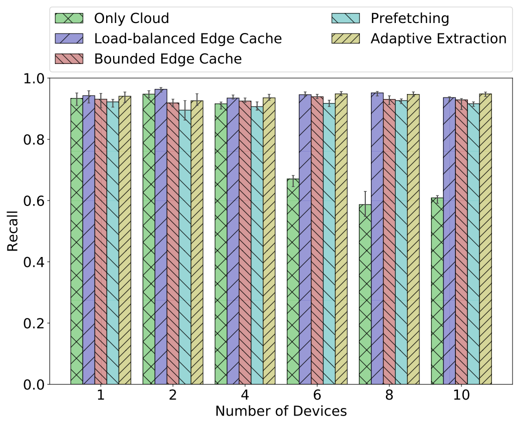
(b) Edge CPU Usage.

Figure 6.5: The effect of varying number of visitors on the CPU usage in different system configurations.

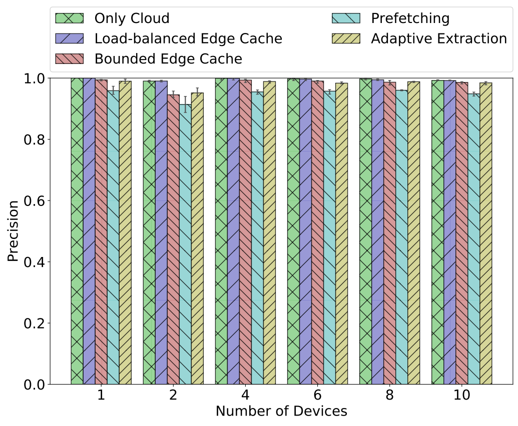
and as the number of devices increase, the compute resources get saturated, as seen in Figure 6.5a. This leads to queuing up of images and further delays.

We see a similar trend in the edge-cache-based system configurations as well. Both edge cache configurations have low response times initially, but by 8 and 10 devices, it has significantly increased. The load-balancing edge cache delivers the fastest response times for low number of devices. The cause is similar here as well. Initially, the edge cache can handle the rate of requests it is receiving and can respond in time, rarely offloading to the cloud. The load-balancing edge cache fares better here since the bounded edge cache is conservative, and offloads to the cloud, as seen by increased cloud CPU usage in Figure 6.5a, even when the edge server's CPU is not fully utilized (Figure 6.5b). For high number of devices, the bounded edge cache fares better since it utilizes less resources, while delivering similar response times as the load-balancing edge cache.

Of all the configurations, the on-device computation configurations fare the best (low infrastructure resource usage, low response times) when number of devices is high, with the adaptive extraction configuration being consistently fast. When the number of devices is low, the edge cache and the wireless network are relatively free, and they can respond to requests quickly, leading to low response times. At higher number of devices, since the edge cache will get congested and hence lead to delays, the system decides to use on-device computation, to reduce the number of requests that get offloaded, as is evident in the lower CPU and bandwidth usage for these



(a) Recall



(b) Precision

Figure 6.6: The effect of varying number of visitors on accuracy in different system configurations.

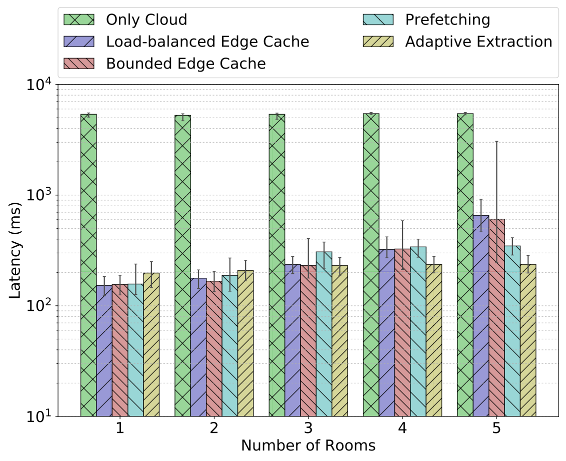
configurations. Of the two on-device configurations, adaptive feature extraction consistently performs better. Not only is its on-device compute time lower, but this also leads to lesser requests being offloaded to the edge cache.

For most configurations, both, recall and precision, are consistently high. Except for the cloud configuration for high number of devices. The recall suffers in this case because the cloud starts to drop requests in an attempt to drain requests that have been queued for more than five seconds. This means many requests are returned as unrecognized and thus it leads to a drop in recall.

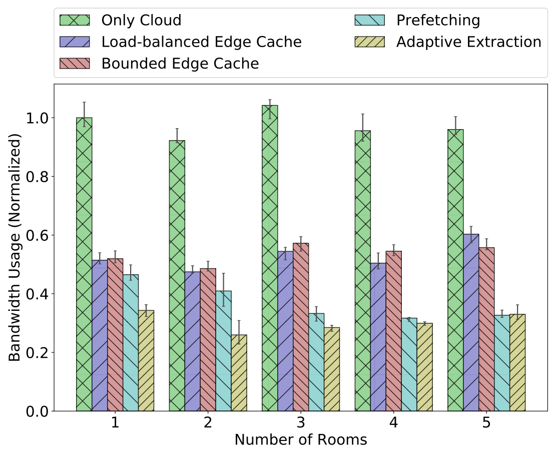
In summary, for any number of devices, having an edge cache is necessary for low response time. Relying only on the cloud is slow even for one or two devices. For few devices, edge caches deliver low response times and lower the cloud and network resource usage. However, to handle requests form a large number of devices, on-device caching and recognition, as implemented in Cachier, needs to be used to provide low response times, low resource usage, and high accuracy.

6.4 Impact of Object Density

This experiment evaluates the effect of object density in the area that an edge server serves, i.e. how does changing the number of recognizable objects in the area served by the edge server affect performance and other metrics. This is useful to know because it can provide insight into how densely edge servers will need to be deployed. Thus in this experiment, we control the number of sections that are served under a single edge server. All the other variables are kept at the baseline

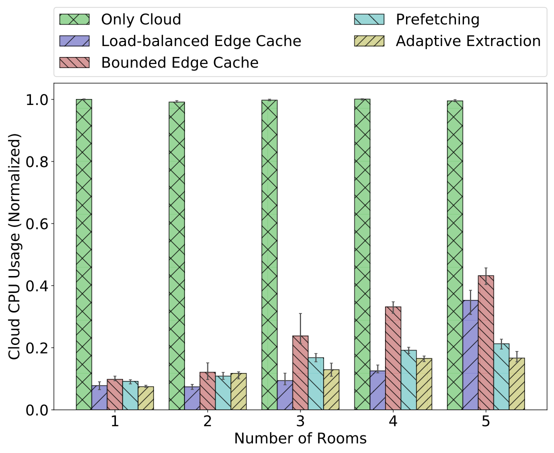


(a) Latency or response time.

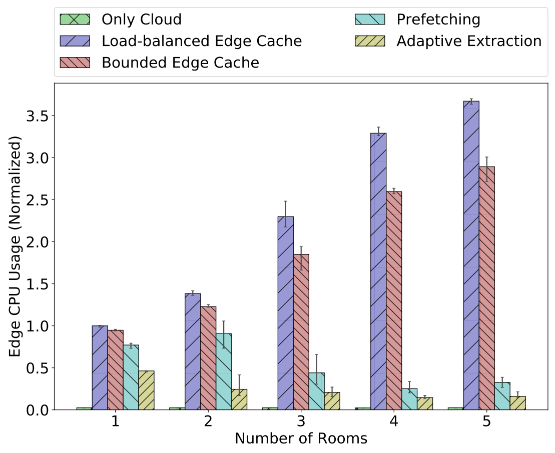


(b) Bandwidth utilization.

Figure 6.7: The effect of varying number of sections on the different system configurations.



(a) Cloud CPU Usage.



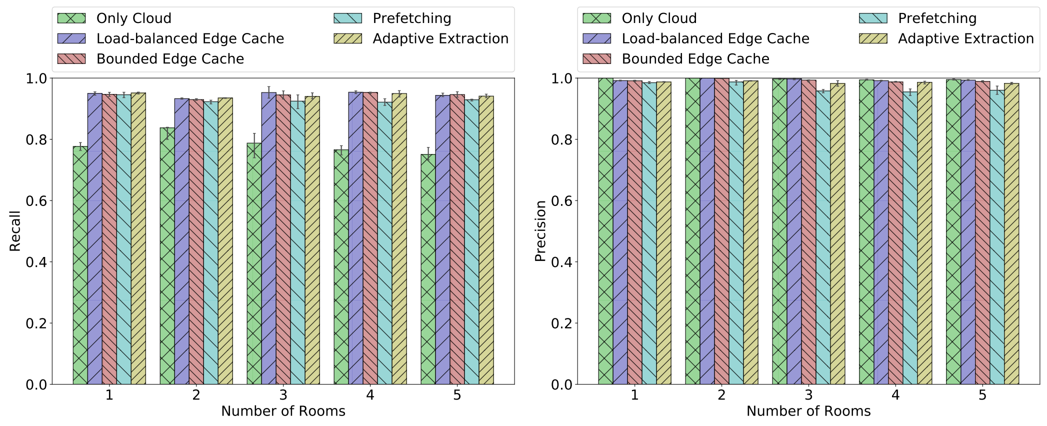
(b) Edge CPU Usage.

Figure 6.8: The effect of varying number of sections on the CPU usage in different system configurations.

values.

We see in Figure 6.7, Figure 6.8 and Figure 6.9 that all metrics for the only-cloud configuration is approximately constant, which is expected. The cloud always uses the full model for recognition, hence changing the number of sections does not affect recognition in the cloud.

On the other hand, the metrics for the edge server change significantly. We can a rise in the overall latency in the edge-cache configurations as the number of sections increase. More significant is the rise in the CPU usage in the edge server (Figure 6.8b). As the number of sections



(a) Recall (b) Precision

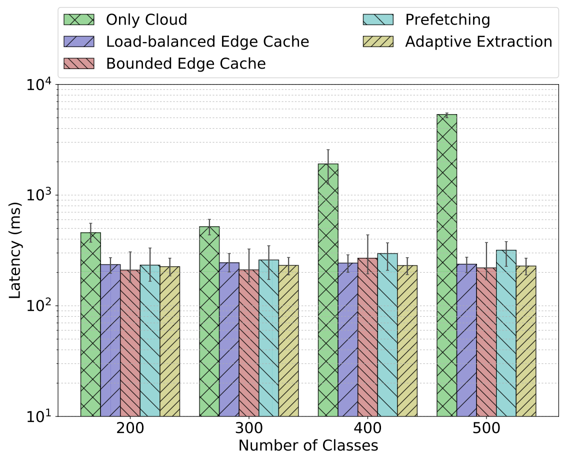
Figure 6.9: The effect of varying number of sections on accuracy in different system configurations.

increase, more objects are covered by a single edge server. This means that it will get requests containing a wider variety of objects and thus will need to cache accordingly. This increases the size of the cached model on the edge server, thus leading to higher compute time. As the compute time increases, the load balancing mechanisms in both edge cache configurations kicks in and they start offloading more requests to the cloud to make use of the cloud's resources.

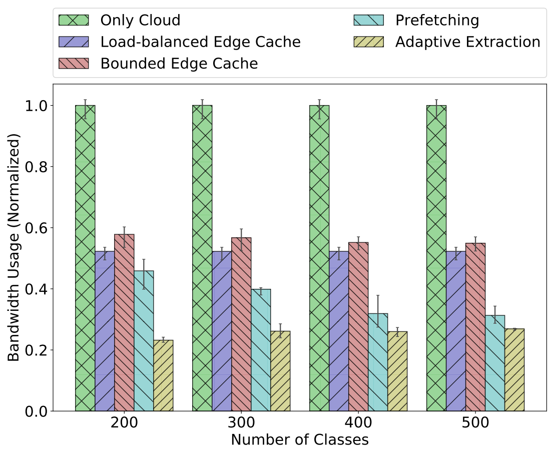
This behavior is in contrast with the behavior of on-device configurations. Here we see that with increase in number of sections, the devices use lesser of the edge resources, evidenced by the drop in bandwidth utilization (Figure 6.7b) and edge CPU usage (Figure 6.8b). But these configurations still remain faster than the edge server and cloud configurations. With low number of sections, compute time at the edge is low, hence the miss latency for the device cache is low. This favors having a really small device cache and offloading more requests. But when the compute time at edge increases due to the increase in the number of sections, Cachier reacts by growing the device cache (and thus model) to enable more on-device recognition, and offload lesser number of requests to the edge cache.

For most configurations, both, recall and precision, are consistently high. The cloud configuration recall is consistent, but low because of the request drop mechanism explained earlier.

In summary, having an edge cache allows a recognition system to leverage locality and drive down response time, compared to an only cloud system. However, edge caches can get saturated as well, if they serve large areas, e.g. if they are a part of a cellular basestation. By using prefetching and on-device computation, this compute stress on edge caches can be relieved. For



(a) Latency or response time.



(b) Bandwidth utilization.

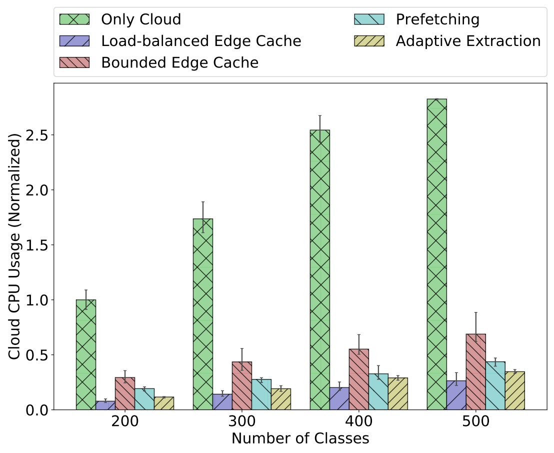
Figure 6.10: The effect of varying number of classes on the different system configurations.

the same large area or number of objects, lesser edge caches would be needed to ensure low latency responses. Thus on-device computation will help systems scale to larger areas.

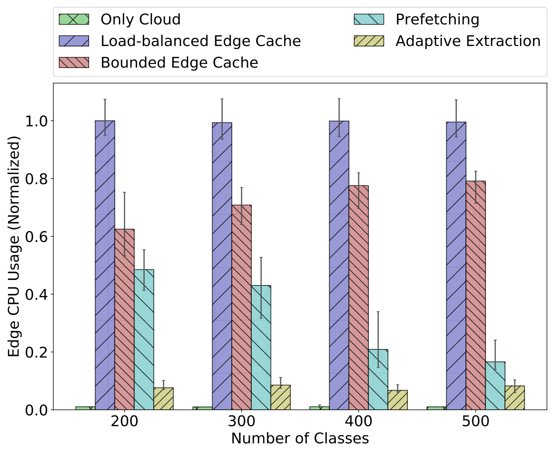
6.5 Impact of Total Number of Object Classes

This experiment evaluates the effect of changing the number of object classes that the application can recognize, or in other words the effect of the complexity of the recognition application. As mentioned earlier, different applications can have different complexities, for example a virtual assistant may be trained to recognize thousands of objects, while a product recognition application may only recognize a specific brand's logo. We want to see how this complexity affects performance. Thus in this experiment, we control the number of object classes that can be recognized by the application. All the other variables are kept at the baseline values.

We see in Figure 6.10 that the response time for cloud configuration increases as the number of classes increase. So does the cloud CPU usage in Figure 6.11. This is expected because as the number of classes increases, the model in the cloud grows, leading to high compute times. The interesting observation here is that the metrics for edge cache stays almost constant. This is because the edge cache only caches those objects which the visitors query for. The visitors can only query for those exhibits which are present in the sections. Since the total number of objects in the sections (44x3) is always lower than the total object classes, changing the object classes does not have an effect on the edge cache. The edge cache can always cache all objects in the section,



(a) Cloud CPU Usage.



(b) Edge CPU Usage.

Figure 6.11: The effect of varying number of classes on the CPU usage in different system configurations.

and avoid going to the cloud at all (except for recognition misses).

The on-device configurations have a similar trend as the previous experiment - with increase in number of classes, the devices use lesser of the edge resources, especially for the prefetching configuration, shown by the drop in bandwidth utilization (Figure 6.10b) and edge CPU usage (Figure 6.11b). This happens because with higher number of classes, misses in the edge cache are most costly, which affects the on-device configuration when a miss happens in the device cache. To reduce such misses, Cachier increases the device-cache size, so as to have fewer misses.

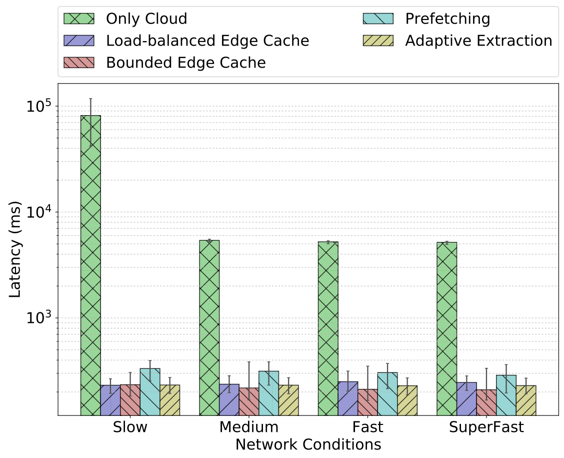
For most configurations, both, recall and precision, are consistently high. The cloud configuration recall is consistent, but drops for the highest number of classes because of the request drop mechanism explained earlier.

6.6 Impact of Network Conditions

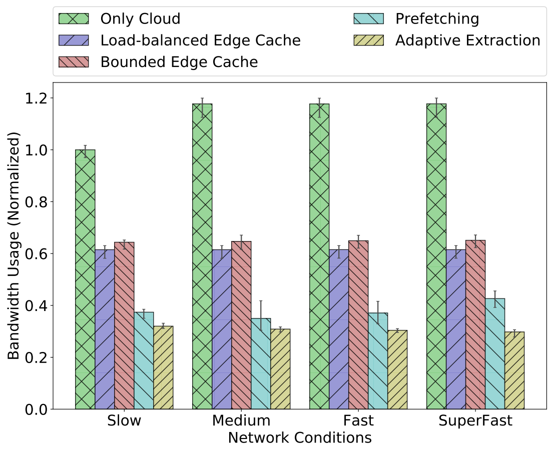
One of the factors motivating edge computing is that the network can become congested and lead to delays. This experiment evaluates how network conditions affect performance and other metrics of the system. We only change the wired portion of the network, i.e. between the edge server and the cloud. Thus in this experiment, we control the network conditions, i.e. the RTT and bandwidth. All the other variables are kept at the baseline values. The labels on the x-axis in Figure 6.12, Figure 6.13 and Figure 6.14 correspond the values in Table 6.3. These values were

Slow	Medium
40ms RTT, 2 Mbps↑, 10 Mbps↓,	30ms RTT, 5 Mbps↑, 10 Mbps↓
Fast	**SuperFast**
20ms RTT, 10 Mbps↑, 20 Mbps↓,	10ms RTT, 20 Mbps↑, 40 Mbps↓

Table 6.3: Network condition values



(a) Latency or response time.



(b) Bandwidth utilization.

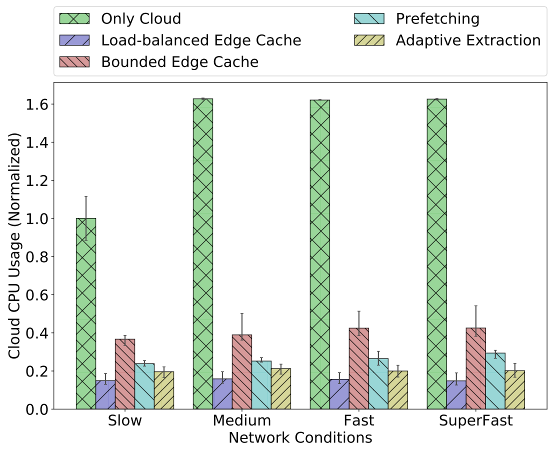
Figure 6.12: The effect of varying network conditions on the different system configurations.

taken from measurements done in [38].

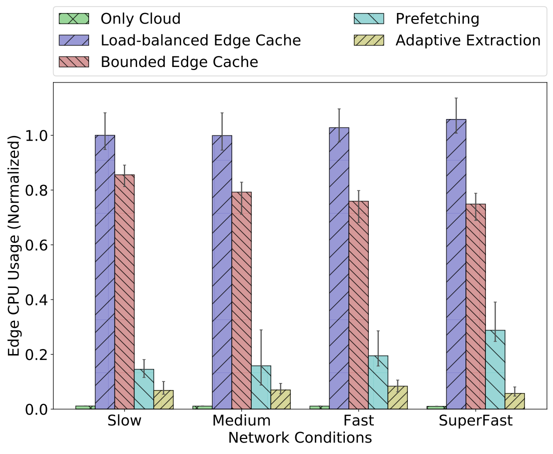
We see in Figure 6.12 that the only cloud configuration has very high response times when the network is really slow. This is expected behavior. However, as the network gets faster, the response time does not decrease. This is because the bottleneck has shifted from network to compute.

Interestingly, we see that the edge server and on-device configuration, both are largely unaffected. Their metrics are relatively constant, similar to the previous experiment's results. This happens because the edge cache serves most requests from its local cache. So changing the network between the edge cache and the cloud does not significantly influence the response times for these configurations.

In summary, an edge cache can shield devices from poor network conditions or congestion in the backhaul, and from any changes in compute time in the cloud, thus ensuring low response times and high accuracy.

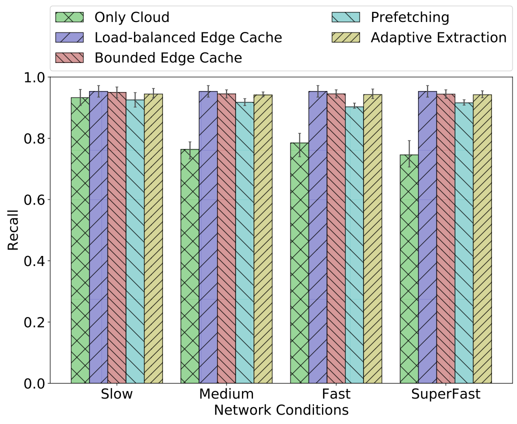


(a) Cloud CPU Usage.

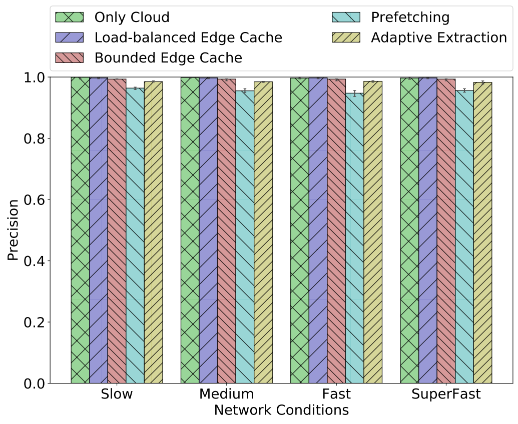


(b) Edge CPU Usage.

Figure 6.13: The effect of varying network conditions on the CPU usage in different system configurations.



(a) Recall



(b) Precision

Figure 6.14: The effect of varying network conditions on accuracy in different system configurations.

	Only Cloud	Edge Cache	Device Cache
Number of Devices	$\leq 600ms$ for 1 Device	$\leq 300ms$ for ≤ 7 Devices	$\leq 300ms$ for ≥ 1 Device
Object Density	No Effect	$\leq 300ms$ for ≤ 150 Objects	$\leq 300ms$ for ≥ 50 Objects
Number of Classes	$\leq 600ms$ for ≤ 300 Classes	No Effect	No Effect
Network Conditions	$\geq 5s$ for all conditions	No Effect	No Effect

Table 6.4: Summary of insights from the evaluation.

6.7 Summarizing Effect of Scale

In all the previous experiments we see that edge caches are essential to a recognition system. Even under low loads offloading requests directly to the cloud is not fast enough. Moreover, we see that the full system, with on-device computation, prefetching and adaptive extraction, performs the best or at par with only edge-cache configurations in most scenarios. It delivers faster responses, uses lesser infrastructure resources and is accurate. This especially holds for high loads, represented by high number of devices and high number of objects. In these scenarios the full system scales much better than the one without on-device computation. If high loads are not expected in a system, then on-device computation can be avoided - in fact, the system will automatically not carry out computations on the device and directly offload to the edge server.

Table 6.4 summarizes the benefits of each component of Cachier. It shows what the upper bound on the latency might be under the given conditions.

6.8 Evaluating the Key-Value Store

The goal of the experiments was to evaluate the K-V store and compare it with Kademlia [61], a popular DHT, for a group of wireless mobile devices. The metrics for comparison were end-to-end latency for getting the value of a specified key. This measures the responsiveness of the system.

Setup. Our experiments were setup to emulate a group of 20 nearby users. Each user's mobile device was emulated by a virtual machine configured with 1 CPU and 2GB RAM. These 20 virtual mobile devices were connected to a simulated access point, over a simulated wireless-medium, using NS3 [71].

Procedure. A different trace of requests was generated for each mobile device. During the experiment, the requests made by a virtual mobile device was in accordance with its assigned trace. The procedure followed by each device for each experiment:

1. Discover all devices and stabilize

2. Store keys from the user-trace in key-value store
3. Issue get requests as per its user-trace

Note, the metrics are measured during the get-requests phase only.

6.8.1 Modifying Kademlia

Bootstrapping. Kademlia, and any DHT in general, starts up using at least one “well-known" node which every new node contacts to join the DHT. Having such a node is infeasible in the scenarios that our K-V store is meant for. Hence, the same discovery layer as the k-v store is used in Kademlia for the experiments. The discovered nodes are treated as neighbour nodes and the normal Kademlia protocol for joining takes over.

Bucket size. Kademlia has a tunable parameter, *k*, which controls the size of each bucket. A bucket is a list of neighboring nodes at an equal “distance" from the given node. Each node contains multiple buckets, each for a fixed distance from this node[1]. To make a fair comparison we set *k* to 20. This way every node in the network knows every other node in the network and hence effectively it would not need multiple hops to reach a specific node.

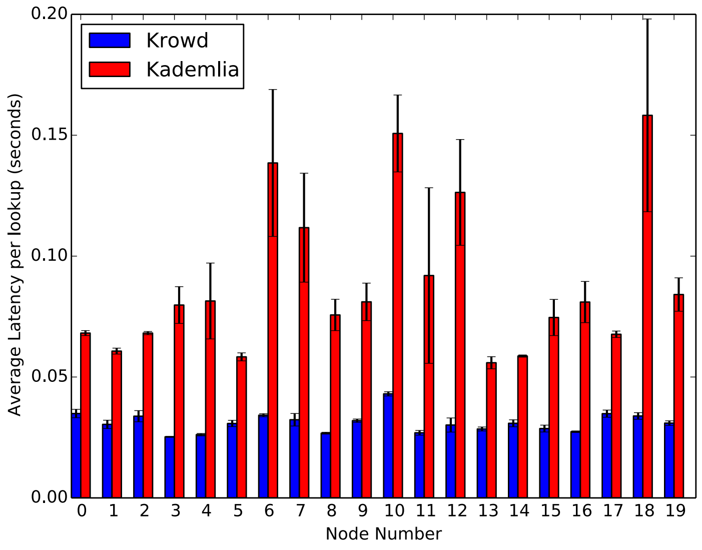


Figure 6.15: Comparing lookup latencies in our K-V Store and Kademlia.

[1]For further details about the distance metric and how it is established, please see [61]

6.8.2 Insights

Latency. In Figure 6.15 we see that for each node, our K-V store has lower end-to-end latency than Kademlia. The latency is measured as the time taken starting from issuing a get request to when a value is returned to the application. The graph shows average latency per request. Note, not only is our K-V store much faster but is also consistent across all nodes, approximately taking the same time per request on each node, compared to Kademlia. This is because of multiple reasons. First, Kademlia employs heuristics for accelerating lookups such as parallel lookups and storing fetched key-values in nearby nodes for faster subsequent lookups. On a local, wireless medium these heuristics cause harm - every additional lookup or store blocks other requests/lookups. A second reason is the periodic pings used by Kademlia for liveness. This again causes interference with other operations on the wireless medium. Our K-V store, instead, uses the broadcast property of the wireless medium in its discovery and presence protocol. It does not need to ping each node separately to check if it is still available.

DHTs were designed for the Internet and wide-area wired networks and a number of heuristics were used to lower latency of requests across the Internet. However in a local, wireless network, these heuristics are ineffective and in fact degrade performance.

Since our system is designed specifically for a local, wireless network from the ground up, we make sure there is no duplication of effort. It uses the network's properties and is thus faster and more efficient than systems designed for the Internet and wired networks.

Chapter 7

Conclusion and Future Work

Increasingly, applications on edge devices such as mobile smartphones, drones, cars, and other IoT devices are relying on machine learning and recognition techniques to provide interactive and intelligent functionality. Given the complexity of these techniques, and resource constrained nature of edge devices, applications rely on offloading compute intensive recognition tasks to the cloud. This has also lead to the rise of cloud-based recognition services. A number of these applications require immediate responses since they may be taking actions based on the response, or interacting with the user. However, using a cloud-based service involves sending captured images to remote servers across the Internet, which leads to slower responses. With the rising numbers of edge devices, both, the network and such centralized cloud-based solutions, are likely to be under stress, and lead to further slower responses.

To address this challenge in this dissertation, we created a distributed caching framework to use the resources on the cloud, edge servers and devices more effeciently. This caching framework leverages the spatiotemporal locality of requests from nearby users to carry out recognition closer to/on the edge devices to minimize response time. We developed a model for expected latency in an image-recognition cache and showed how to incorporate the effects of compute-intensive recognition algorithms in this model. We then used this model as a guiding formulation to minimize expected latency by dynamically adjusting its cache size. This was first utilized for edge servers that serve multiple devices in their respective areas. Then it was applied to caches on devices. We introduced prefetching for device caches as a way to avoid compulsory misses. We further optimized the feature extraction phase and made it dynamic so that it can react to the cache size on the device. We then finally presented a distributed key-value store created on nearby devices themselves, to leverage locality, and allow devices to cooperate directly with each other. We demonstrated the effectiveness of this caching approach by implementing an end-to-end system, along with an application and then evaluating each component of the system. We used real world datasets to drive the evaluation. The main contributions of this caching framework, Cachier, is as follows:

Caching for Recognition. Caching for recognition essentially means pushing parts of the cloud-based model being used for recognition to the edge, i.e. onto edge servers and devices. Doing so will push the computation towards the edge, and reduce the number of times requests need to traverse the Internet and thus reduce response time. To achieve this, we introduced a model for a recognition cache. Recognition is complex and its compute time depends on the size of the cache, or rather the model being used for recognition in the cache. Moreover, the misses in the cache can now also happen when the recognition algorithm fails to recognize the object in the request. Both these factors are taken into account in building the model. This model serves as a mechanism to estimate latency in the different parts of Cachier where a recognition cache is implemented.

Recognition cache at the edge. Edge computing has been proposed as a method to address the challenge of recognition applications' low-latency requirements. We proposed to use these edge servers as recognition caches. Instead of deciding which parts of the application to execute at the edge, Cachier dynamically decides which subset of the recognition classes to be included in the trained model at the edge. Cachier makes use of the formulation developed earlier to make this decision. To achieve this, we presented the different estimation methods involved in estimating the formulation and then finding the cache size, k, that minimizes it. We showed that once this is found, a model for the most frequent k items is cached at the edge server, which is used to recognize future requests. By making the cache size adapt to dynamic conditions, Cachier reduces the response time.

Prefetching for on-device recognition. Although having a recognition cache at the edge server reduces response times by not going to the cloud as often, requests still need to incur latency due to the wireless network when they try to reach the edge server. Moreover, the edge server can become a bottleneck if there are multiple users. To decrease response time further, we showed how caching can be used for efficient, on-device recognition. We introduced the concept of prefetching models on to the device to avoid compulsory misses. By having the devices collaborate with the edge server, the edge server predicts which object the user/device might see next. Cachier uses this information, along with the caching model to decide how many such predicted models to prefetch onto the device. This takes into account the resources available on the device.

Adaptive feature extraction. By prefetching models into the device cache and recognizing objects on the device, we showed that dependence on the edge server can be decreased. However, edge devices are resource constrained, and recognition needs to be further optimized to be carried out on devices. We proposed to do this by optimizing the feature extraction phase. To achieve this, the complexity of the feature extraction is adjusted dynamically to match the number of classes in the prefetched model. We analyzed the tradeoffs between accuracy, latency, model size and number of extracted features, and then used this analysis online to decide how many features to extract.

Coordination among nearby devices. In the proposed techniques, Cachier assumes that it has low-latency access to an edge server to carry out a number of mechanisms. However, these edge servers may be deployed deeper in the cellular or Internet network. In such scenarios, it is not feasible to rely on the edge server. Even though prefetching minimizes reliance on the edge server for recognition, it is still required to prefetch the models on to the device. By enabling coordination among nearby devices, our distributed key-value store provides a mechanism for devices to share their downloaded models with other nearby devices. This reduces the number of interactions required with the edge server. Such a distributed key-value also enables other features such as cooperative analytics.

7.1 Open Questions and Future Work

The caching approach introduced in this dissertation, opens up a new way of thinking about machine learning applications and distributed computing at the edge. This also raises open questions about scalable systems that support machine learning applications.

7.1.1 Optimizing Inference on Edge Devices

Research in machine learning, especially deep learning, has mostly focused on algorithms and models for training for high accuracy [24]. Only recently research has started to focus on systems approaches to making this feasible. Since the bottleneck is handling the large datasets, models, and the training time, these new approaches focus on how to accelerate training complex models on distributed clusters and GPUs. Very few efforts have explored the challenge of accelerating inference on edge and embedded devices. [31] have looked at how to use compression and approximation to run complex models on mobile devices that have state-of-the-GPUs. [32] focused on how to tradeoff accuracy, energy and latency when running neural networks on edge devices, and distribute the model across device and cloud accordingly. Since there will be millions of such devices generating data, all of it cannot be uploaded to cloud to be processed. Since this data will likely be processed by machine learning and recognition techniques, it is important these techniques by optimized to run on such devices, to reduce the amount of data that gets offloaded. This will not only help minimize bandwidth bottlenecks, but also reduce remote resource usage, reduce response times and increase privacy.

7.1.2 Distributed Intelligence at the Edge

Given the rise of the number of devices at the edge of the network, as we mentioned earlier, it is necessary to push computations back to the edge. This will enable richer services and faster responses for applications and help systems to manage scale. At the same time, this implies that these edge devices will now be smarter, and run machine learning on-board. Given that these devices will be densely distributed, many should be able to communicate and interact with each other. This then raises the question how can systems or applications benefit when these devices collaborate. In this dissertation we showed one approach to enable such collaboration and provided few techniques that can leverage it. Some research efforts have started to look into it. [59, 72] address distributed learning and push some computation related to training to the edge. This area needs to be explored further, to answer questions such as can devices exchange knowledge or adapted models? Can such distributed learning accelerate training or inference?

7.1.3 Vertically Distributed Systems

In this dissertation, we relied on the new computing paradigm called edge computing. There is another, more generic approach, called fog computing [37]. This approach proposes to distribute computing resources all across the Internet, between the edge and the cloud. So instead of the three levels we considered in this dissertation, fog computing can enable to multiple levels. This means that there can be a rich hierarchy starting at the device, then the edge servers, fog nodes and finally the cloud. Research has typically tackled these entities in pairs - devices and edge servers [36], devices and cloud [34, 33, 73]. We believe that there is an opportunity to look at these different entities as layers in a larger system, engineer and optimize them for end-to-end performance and provide abstractions that make developers oblivious of where there applications will actually run or how they will be partitioned. One could think about it like MapReduce [74] or Spark [75] for vertically distributed systems instead of clusters. That is, ideally we need a system that can automatically place data and computation across this vertically distributed system, taking into account the requirements of the applications and the heterogeneity of the nodes and networks involved. This can ease development of applications and management of systems.

Bibliography

[1] Claudio Martella, Armando Miraglia, Marco Cattani, and Maarten van Steen. Leveraging proximity sensing to mine the behavior of museum visitors. In *Pervasive Computing and Communications (PerCom), 2016 IEEE International Conference on*, pages 1–9. IEEE, 2016. xii, 63, 64

[2] Mobileye. `http://www.mobileye.com/our-technology/`. 1

[3] NavVis. `http://www.navvis.com/products/navigation-app/`. 1, 62

[4] Amazon Flow. `https://www.a9.com/whatwedo/mobile-technology/flow-powered-by-amazon/`. 1, 62

[5] Artbit. `http://www.artbit.com/`. 1

[6] Paul Sawers. Google opens Tango augmented reality platform to museums, starting with Detroit Institute of Arts. `https://venturebeat.com/2017/01/09/google-tango-museums/`, January 2017. 1

[7] Facebook's image recognition can now tell what you're wearing. `http://mashable.com/2017/02/02/facebook-computer-vision-ai-upgrade/`. 1

[8] Snapchat applies for patent to serve ads by recognizing objects in your snaps. `http://www.theverge.com/2016/7/18/12211292/snapchat-object-recognition-advertising`. 1

[9] Google Glass. `https://en.wikipedia.org/wiki/Google_Glass`. 1, 11

[10] Microsoft Hololens. `https://www.microsoft.com/microsoft-hololens/en-us`. 1

[11] Amazon Rekognition. `https://aws.amazon.com/rekognition/`. 2

[12] Microsoft Cognitive Services. https://www.microsoft.com/cognitive-services. 2

[13] Google Cloud Vision. https://cloud.google.com/vision/. 2

[14] IBM Watson. https://www.ibm.com/watson/developercloud/visual-recognition.html. 2

[15] Catchoom. https://catchoom.com/. 2, 58

[16] Vuforia. https://www.vuforia.com/. 2, 58

[17] Marcus K Weldon. *The future X network: a Bell Labs perspective*. Crc Press, 2016. 2, 63

[18] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, March 2017. 2

[19] Jeff Hecht. The bandwidth bottleneck that is throttling the Internet. http://www.nature.com/news/the-bandwidth-bottleneck-that-is-throttling-the-internet-1.20392, August 2016. 2

[20] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science &; Business Media, 2010. 3

[21] David 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. 3, 40

[22] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006. 3, 5, 40

[23] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. IEEE, 2011. 3, 5, 21, 40, 41, 58

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 3, 81

[25] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM. 3

[26] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008. 3

[27] E. Osuna, R. Freund, and F. Girosit. Training support vector machines: an application to face detection. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 130–136, Jun 1997. 3

[28] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 331–344, New York, NY, USA, 2015. ACM. 5, 7, 49, 64

[29] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari 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. 5, 7, 64

[30] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. 5

[31] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016. 5, 82

[32] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016. 5, 82

[33] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *ACM MobiSys*, 2010. 5, 83

[34] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *EuroSys*, 2011. 6, 83

[35] Paramvir Bahl, Richard Y. Han, Li Erran Li, and Mahadev Satyanarayanan. Advancing the state of mobile cloud computing. In *ACM Workshop on Mobile Cloud Computing and Services*, 2012. 6

[36] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009. 6, 17, 83

[37] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM. 6, 7, 17, 82

[38] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 166–176. IEEE, 2013. 6, 74

[39] Utsav Drolia, Katherine Guo, and Priya Narasimhan. Precog: Prefetching for image Recognition applications at the edge (under submission). In *ACM/IEEE Symposium for Edge Computing*, 2017. 8

[40] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Cachier: Edge-caching for recognition applications. In *IEEE International Conference on Distributed Computing Systems*, 2017. 8

[41] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Towards edge-caching for image recognition. In *Smart Edge Computing and Networking (SmartEdge), IEEE International Workshop on*. IEEE, 2017. 8

[42] Utsav Drolia, Katherine Guo, Rajeev Gandhi, and Priya Narasimhan. Edge-caches for vision applications. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 91–92. IEEE, 2016. 8

[43] Utsav Drolia, Nathan Mickulicz, Rajeev Gandhi, and Priya Narasimhan. Krowd: A key-value store for crowded venues. In *Proceedings of the 10th International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '15, pages 20–25, New York, NY, USA, 2015. ACM. 8

[44] Image recognition app TAGR turns the world into a street art gallery. `https://catchoom.com/trust/case-studies/arts-entertainment-education/art-image-recognition-app-tagr/`. 11, 63

[45] PhooDi grocery scanner app turns you into a smart shopperi. `https://catchoom.com/trust/case-studies/food-drink/phoodi-image-recognition-food-scanner-app/`. 11, 62

[46] Wikitude. `http://www.wikitude.com/`. 11, 58

[47] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999. 12

[48] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015. 12, 14

[49] David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2161–2168. Ieee, 2006. 14

[50] Manuel Martinez, Alvaro Collet, and Siddhartha S Srinivasa. Moped: A scalable and low latency object recognition and pose estimation system. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2043–2049. IEEE, 2010. 14

[51] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999. 18

[52] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007. 21, 59

[53] Vijay R Chandrasekhar, David M Chen, Sam S Tsai, Ngai-Man Cheung, Huizhong Chen, Gabriel Takacs, Yuriy Reznik, Ramakrishna Vedantham, Radek Grzeszczuk, Jeff Bach, et al. The stanford mobile visual search data set. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 117–122. ACM, 2011. 21, 41, 63

[54] Xiaoyu Wang, Ming Yang, Timothee Cour, Shenghuo Zhu, Kai Yu, and Tony X Han. Contextual weighting for vocabulary tree based image retrieval. In *2011 International Conference on Computer Vision*, pages 209–216. IEEE, 2011. 21

[55] Venkata N Padmanabhan and Jeffrey C Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996. 33

[56] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006. 41

[57] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2548–2555, Washington, DC, USA, 2011. IEEE Computer Society. 41

[58] Gabriel Brown. Mobile edge computing use cases & deployment options, July 2016. 47, 49

[59] Dawei Li, Theodoros Salonidis, Nirmit V Desai, and Mooi Choo Chuah. Deepcham: Collaborative edge-mediated adaptive deep learning for mobile object recognition. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 64–76. IEEE, 2016. 49, 82

[60] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001. 51, 53

[61] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*. Springer, 2002. 51, 53, 76, 77

[62] David G Thaler and Chinya V Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 1998. 53

[63] ZeroMQ. `http://zeromq.org/`. 58

[64] Google protocol buffers. `http://developers.google.com/protocol-buffers/`. 58

[65] OpenCV Library. http://opencv.org/. 58

[66] Samsung. Samsung bixby. http://www.samsung.com/us/explore/bixby/overview/. Accessed on 06.22.2017. 62

[67] Sarah Perez. Google lens will let smartphone cameras understand what they see and take action. https://techcrunch.com/2017/05/17/google-lens-will-let-smartphone-cameras-understand-what-they-see-and-take-action/, May 2017. 62

[68] Randy Rieland. Augmented Reality Livens up Museums. http://www.smithsonianmag.com/innovation/augmented-reality-livens-up-museums-22323417/, August 2012. 63

[69] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. The case for offload shaping. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 51–56, New York, NY, USA, 2015. ACM. 64

[70] netem. https://wiki.linuxfoundation.org/networking/netem. 66

[71] Henderson, Thomas R et al. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008. 76

[72] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016. 82

[73] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *ACM MobiSys*, 2011. 83

[74] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. 83

[75] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX. 83