DESIGNING NOVEL MOBILE SYSTEMS BY EXPLOITING SENSING, USER CONTEXT, AND CROWDSOURCING

A Dissertation Presented

by

TINGXIN YAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2012

Department of Computer Science

DESIGNING NOVEL MOBILE SYSTEMS BY EXPLOITING SENSING, USER CONTEXT, AND CROWDSOURCING

A Dissertation Presented

by

TINGXIN YAN

Approved as to style and content by:

Deepak Ganesan, Chair

Donald Towsley, Member

Prashant Shenoy, Member

Mark Corner, Member

Weibo Gong, Member

Lori A. Clarke, Department Chair
Department of Computer Science

To Yanli and Suri

ACKNOWLEDGMENTS

I would like to express my heartiest appreciation to many people who have helped me a lot during the past a few years. This thesis is impossible without the help of them.

First, I would like to thank my advisor, Deepak Ganesan. He has been a perfect role model to me, for the sake of his consistent technical curiosity and exquisite research vision, strong will of pursuing perfection, and great skills of planning and execution. His advices and supports are invaluable for me. I thank him for encouraging me to aim high and persevere my goals, for spending enormous time with me to crystallize ideas and build prototyping systems, and for helping me improve my writing and presenting skills.

I would also express my thanks to my committee members — Don Towsley, Mark Corner, Prashant Shenoy, and Weibo Gong. Don taught me critical modeling and system evaluation skills. Mark pointed me to the research domain of mobile crowdsourcing. The advices from Prashant and Weibo on my research have also helped me to improve this thesis dramatically.

I would like to thank all my labmates and colleagues in UMASS, including Peter Desnoyers, Timothy Wood, Ming Li, Dan Xie, Devesh Agrawal, Vikas Kumar, Upendra Sharma, Jeremy Gummeson, Moaj Musthag, Abhinav Parate, Pengyu Zhang, and many others. I have shared countless of joyful moments with them fulfilled with brainstorming, team working, discussing, and laughing. I will always miss the time being a graduate student in the room of CS214.

PhD life was not always about coding and running experiments in the lab. I was fortunate to know lots of exceptionally smart and easy-going friends here in UMASS, including Wentian Lu, Xiaobing Xue, Yimin Wu, Chang Wang, Bo An, Hong Yuan, and many others. I have learned a lot from them and I will miss the time being with them.

Lastly and most importantly, I would like to say thank you to my beloved family. You are the source of my inspiration, courage, and perseverance. Without your support, this thesis could be an impossible mission. I appreciate the understanding from my parents, especially their support to my decision of studying abroad. I owe immense gratitude to my wife, Yanli Zhao, for her steady support and encouragements. Thank you for all that you have done for me. I am so delighted to dedicate this thesis to our baby girl, Suri.

ABSTRACT

DESIGNING NOVEL MOBILE SYSTEMS BY EXPLOITING SENSING, USER CONTEXT, AND CROWDSOURCING

SEPTEMBER 2012

TINGXIN YAN

B.Sc., NANJING UNIVERSITY

M.Sc., CHINESE ACADEMY OF SCIENCES

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Deepak Ganesan

With the proliferation of sensor-enabled smartphones, significant attention has been attracted to develop sensing-driven mobile systems. Current research on sensing-driven mobile systems can be classified into two categories, based on the purpose of sensing. In the first category, smartphones are used to sense personal context information, such as locations, activities, and daily habits to enable applications such as location-aware systems and virtual reality systems. In the second category, smartphones are exploited to collect sensing data of the physical world and enable applications including traffic monitoring, environmental monitoring, and others.

As smartphones become blossomed in popularity and ubiquity, new problems have emerged in both categories of mobile sensing systems. In this thesis, we investigate three core challenges by answering the following fundamental questions: first, how can we utilize user context to improve the operating system performance? Second, how can we process sensing data, especially images, with high accuracy? Third, how can we enable distributed sensing while satisfy resource constraints of smartphones?

The first part of this thesis studies how to exploit user context to improve the responsiveness of mobile operating systems. We propose a context-aware application-preloading engine named FALCON. The core of FALCON is a decision engine that learns application usage patterns of mobile users and preloads applications ahead of time to improve the responsiveness of mobile OS. Compared with other approaches such as caching schemes like Least Recently Used (LRU), Falcon improves the application responsiveness by two times.

The second part of this thesis focuses on image search for mobile phones. We first explore how to improve image search accuracy on centralized servers, and propose an image search engine named CrowdSearch. The core idea of CrowdSearch is to incorporate crowdsourced human validation into the system for removing erroneous results from automated image search engines, while still provide realtime response for mobile users. Compared with existing automated image search engines, CrowdSearch achieves over 95% accuracy consistently across multiple categories of images with response time in a minute.

We then extend image search to distributed mobile phones, and emphasis resource constraint problems, especially on energy and bandwidth. We propose a distributed image search system named SenSearch, which turns smartphones into micro image search engines. Images are collected, indexed, and transmitted using compact features that are two magnitudes smaller than their raw format. SenSearch improves the energy and bandwidth cost by five times compared with centralized image search engines.

TABLE OF CONTENTS

Page

CHAPTER

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1
INTRODUCTION

1.1 Overview

Twenty years ago, Mark Weiser envisioned that computation in the 21st century would be "conveniently at hand and distributed throughout our everyday environment" [128]. In the past decade, we have witnessed that his vision is realized due to the proliferation of smartphones. With ubiquitous computation, communication, and sensing capabilities, smartphones have become the center of digital life thereby influencing millions of people. One significant trend is that smartphones have begun to replace personal computers as the major computation platform. For instance, a market survey in 2011 indicated that mobile Internet will surpass traditional personal computer-based Internet access in 2014 [104]. Some other studies also show that people spend several hours per day on mobile phone applications that cover many aspects of our daily life, such as games, social networks, and many others [80].

A significant difference that distinguishes smartphones from other computational devices is their rich set of on-board sensors. Typical smartphone sensors include GPS, camera, microphone, proximity, ambient light, digital compass, accelerometer, and gyroscope. Although these sensors were initially included in phones to enhance user interactions, the sensing capability of smartphones can be exploited in many other ways, and has been exploited by a large spectrum of mobile systems to enable novel applications, including traffic monitoring [6, 126], environmental monitoring [82, 48], community healthcare [25], augmented reality [94, 9], and others.

Sensing-driven mobile systems can be divided into two major categories, based on the purpose of sensing. The first category of mobile systems is called *context-aware mobile systems*, which take advantage of the sensing capability of smartphones to obtain *context information* about mobile users. Typical personal context information includes location, activities of a user, social context, and temporal patterns of daily habits, such as time of day patterns. Such personal context information can be valuable in improving the user experience of mobile systems. For example, a restaurant recommendation application can automatically adjust its search radius depending on whether a user is on foot, cycling, or driving, which can be inferred from GPS and accelerometer [35]. Alternately, a social network application can enable message alerts at an interruptible moment such as when she is not engaged in conversation, which can be inferred from ambient noise sensed by microphones [91].

Besides sensing personal context, smartphones can be used to sense the physical world as well. Millions of mobile users carry smartphones, and can potentially provide a broader coverage of the physical world than any other deployed sensing system. Such sensor information collected by mobile users can potentially be utilized to facilitate large scale sensing applications, which in turn can have significant societal impact. For example, a large number of mobile phone users contribute traffic information to enable realtime traffic maps, and this mechanism has been adopted by several navigation software, including the recently released Apple Map in iOS6 [6]. Another example is UCLA's PEIR project [82], which enables mobile users to report personalized environmental impact such as carbon emissions through smartphone-based systems.

Although smartphones have the potential to be used for sense contextual information of mobile users as well as the physical world, new problems have emerged in terms of how to fully exploit the sensing capability of smartphones for novel applications. Specifically, we tackle three major challenges in this thesis as follows:

- *How can we exploit the contextual sensing information to improve the system performance of smartphones?* In other words, we explore the correlation between personal context and smartphone operating systems, and take advantage of the context information to improve the performance of key components of smartphone operating systems.
- *How can we exploit crowdsourcing to better process the sensing information collected by smartphones?* Especially, we explore the problem of improving the accuracy of image search for mobile phones by exploiting crowdsourcing, and we tackle the challenge of providing an accurate image search with low latency and low monetary cost.
- *How can we exploit multimedia sensing, in particular images, to enable distributed mobile applications?* Specifically, we explore the opportunity of searching across images that are collected by smartphones, and we tackle the challenge of realizing image search for distributed smartphones with better energy efficiency.

In the rest of this chapter, we will introduce the background, motivation, and core ideas of our work on each of these problems. After that, we conclude this chapter by summarizing our contributions.

1.2 Context-aware Mobile App Preloading

Context-aware computing is a mobile computing paradigm where applications and systems can discover and take advantage of the contextual information of mobile users, including user location, user activities, time of day, social contacts, and nearby people and devices [19]. Smartphones are a natural platform for context-aware computing given their sensing and computation capability. Within the past decade, there have been numerous context-aware applications that take advantage of the sensing capability of smartphones to sense the context of mobile users and use the

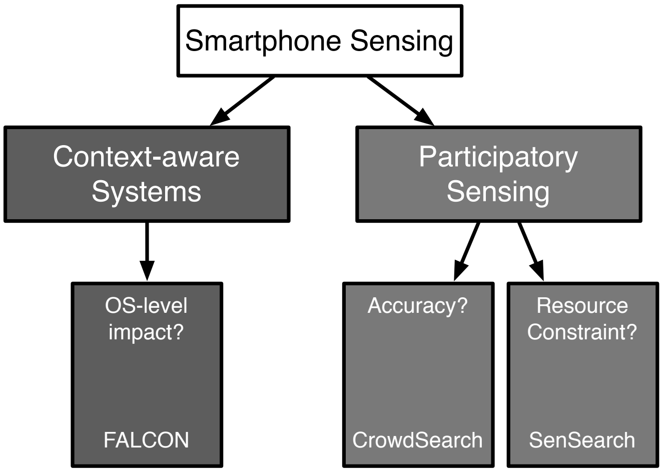


Figure 1.1. The mobile systems proposed in this thesis explore the challenges of exploiting smartphone sensing for 1) improving OS performance, 2) accurately processing sensing data, and 3) energy- and bandwidth-efficiently processing sensing data.

context for various purposes. For example, social applications exploit user locations to enable location-aware sharing [32, 39, 77, 103, 110, 131]; user activities and gestures can be inferred from motion sensors such as accelerometer, and can be used for various applications such as inferring daily activity patterns and detecting parking events [4, 64, 75, 109, 124]; ambient noise can be used for detecting surrounding environment [63], and GPS traces are used for realtime traffic applications [38, 45, 72, 127].

While user context has been widely used in a variety of mobile applications, we argue that context can be exploited to improve the performance of mobile operating systems as well. For example, mobile users may use different sets of applications with different activities — workout apps when running, and map apps when driving. The memory management module of an operating system can exploit the user activity information to decide which set of applications to maintain in the memory. Another example is the use of logical locations to provide hints to energy policy module of mobile phones — when users are in a mall or riding a bus and the battery level is low, it suggests that the next recharge opportunity may be some time off, which in turn could lead to more conservative energy policies such as disabling energy-expensive services such as automatic email fetching.

In this thesis, we focus on a specific problem in mobile operating systems — *how can we improve the responsiveness of mobile applications by reducing their loading time*. Interacting with mobile applications can be surprisingly sluggish on existing smartphones. Applications such as games can take dozens of seconds to launch. Even for relatively simple applications such as weather reporting apps, the launch delay can take 10 seconds or more. Other applications, such as email clients, suffer from slow network content fetching over the network. While appropriate for a restricted set of interactions, stale content fails to engage the user in the same way that the latest fresh content can. When considering the full launch time starting from touch until the latest content is displayed, applications with asynchronously loaded content can also exhibit severe load times of up to a dozen seconds.

The most common approach to decrease launch time is to cache applications in memory. For instance, LRU (Least Recently Used) [100] evicts applications based on when an application is used, and only keeps the most recently used ones in memory. Caching schemes are implemented by iOS, Android and Windows Phones, yet suffer from several drawbacks. First, some applications, especially games, demand large amounts of memory, thereby reduce the memory real estate for other applications. Second, caching does not address the issue that content may become stale by the time the user interacts with the application.

Compared with caching-based schemes, active preloading can significantly decrease the loading delay while avoid the drawbacks that caching schemes suffer. When an application is correctly preloaded a few minutes ahead of its actual usage, the benefit comes from both reduced program loading time from flash into memory, and reduced network content fetching time.

On the other hand, application usage on smartphones has strong correlations with mobile context, such as location information. For example, for a particular user, Angry Birds usage may be much more likely at home, whereas Email and Calendar usage may be much more likely at work. Or, a user may habitually check Facebook after receiving an SMS when at school.

In this thesis, we explore the design of a context-aware mobile operating system named FALCON, where user context can be used to predict which application to preload. Our core idea is to learn the predictive pattern of app usage of mobile users, and prelaunch applications based on context signals and user access patterns that we learned. Prelaunching entails both preloading applications in memory and app-specific fresh content fetching. Our contribution are two-folds. First, we learn the predictive usage patterns and their correlation to user context. Second, we show that by taking advantage of the mobile context, especially spatial and temporal patterns, the app loading performance can be improved significantly. Our evaluation indicates that the responsiveness of mobile applications can be improved by two times compared with caching schemes such as LRU.

1.3 Crowdsourcing-based Accurate Image Search for Smartphones

Besides sensing personal context, sensor-enabled smartphones is an ideal platform for sensing the physical world as well. With the sheer volume of smartphones and the rich set of on-board sensors, smartphones can be potentially used for a range of large-scale sensing applications. For instance, mobile users can be leveraged to provide realtime vehicle speed information, and such information can be used to generate traffic condition maps to provide realtime traffic guidance to other drivers [6].

Among all the sensing data that smartphones are able to collect, image content is particularly interesting for several reasons. First, camera is a ubiquitous sensor on smartphones, and perhaps featured phones as well. Second, large volume of images are captured and shared by mobile users, and these images provide a substantial coverage of the physical world.

Many mobile applications are designed to take advantage of the volume of images captured through mobile phones. Augmented reality applications infer landmarks and public service signs such as subway stations from camera scenes [1, 18, 51, 102]. Social apps allow users to share images with tags such as logical locations [32, 34, 36, 110], and allow users to recognize people in images and tag images with friends names [32]. Face recognition is also used to automatically recognize people in pictures and tag images accordingly [49, 88].

Image search is a new paradigm that utilizes smartphone images as search queries instead of text to facilitate search. A typical use case of mobile image search is as follows: a tourist is visiting the Golden Gate bridge of San Francisco for the first time, and the tourist wants to know more about the bridge. Instead of typing a text query such as "what is the name of the famous red bridge in SF" on the small screen of a smartphone, she can take an image of the bridge and send it to an image search engine, and get information about the bridge back. One example image search engine is Google Goggles [41], which allow mobile phone uses to search through images include landmarks, textbooks, and arts by using images from smartphones.

Image search greatly extends the scope of search functionality of smartphones, but it also poses the following challenge: *how to build an image search engine that provides accurate search results?* It is a hard problem for two reasons. First, existing image processing techniques are limited by a series of factors such as lighting, texture, type of features, image quality, and others. Therefore, it is hard for automated image recognition techniques to accurately interpret the content of images. As a result, even state-of-art image search systems, such as Google Goggle [41], acknowledge that they only work with certain categories of images, such as landmarks, textbooks, and art works. Second, scrolling through multiple pages of search results is inconvenient on a small screen of smartphones. This makes it important for search to be precise and generate few erroneous results.

Although computational methods have limitations in terms of processing images accurately, humans are generally very accurate in interpreting the image content. In recent years, online crowdsourcing services, such as Amazon Mechanical Turk (AMT) [53], have begun to utilized a large online workforce to distinguish image content. For instance, people in AMT can be asked to provide tags for images, or validate if an image is correctly tagged, or identify whether several images contain same object or not.

Inspired by the power of crowdsourcing in processing image content, we ask the following question in this thesis: *can we combine the advantages of automated image search and crowdsourcing to provide accurate image search to mobile users, while still providing real time responses?* We design a system called CrowdSearch to answer this question. In CrowdSearch, crowdsourced human validation is used to validate search results from cloud-based image search engines, and return validated results to mobile users. In order to provide realtime response to mobile users, we also need to tackle a set of challenges. For instance, we need to handle the errors caused by human validation, such as human bias and large delay from workers, to ensure accurate human validation results. We also need to schedule human validation tasks to ensure realtime responses, and we need to consider the balance between delay and monetary cost. We show that CrowdSearch achieves over 95% accuracy consistently across multiple categories of images with response time in a minute.

1.4 Energy-efficient Distributed Image Search for Camera Sensor Networks

A natural extension of centralized image search is to enable image search across phones or other camera sensors. In distributed image search, smartphones act as image search engines, which store and search images locally on smartphones rather than centralized servers. Search queries are delivered to each smartphones, and the search results are returned back to the user after local search processing.

The paradigm of distributed image search is especially useful in several application scenarios. For example, bandwidth limits the number of images people can upload to online servers, especially under slow networks such as 3G. With images stored locally and searchable, the scope of image sharing can be enlarged to not only those images uploaded to centralized servers but stored locally on phones. An example of such image sharing is a CNN news article that reported the arrest of a thief whose act of stealing appeared in the background of a family photo [24] stored on phones. Another example is privacy-sensitive image sharing, such as healthcare applications [3] and security applications [44]. In such applications, distributed image search provides an opportunity to store images locally and make them searchable through certain features that cannot expose the content of images. Compared with uploading images to centralized server, distributed image search has the advantages of preserving the privacy of users.

Inspired by recent advances of computer vision where compact features can be use to represent images to enable efficient image search, we design a distributed image search engine named SenSearch, where smartphones are turned into micro image search engines where images are captured, represented by compact features, indexed and searched via transmitting compact features with the cloud. We will show that our approach reduces the energy and bandwidth cost of mobile image search by five times compared with existing approaches where images are sent directly to the cloud.

1.5 Thesis Contributions

We summarize the contributions of this thesis as follows.

We first study the problem of how to incorporate user context into a mobile operating system design by designing FALCON, an application preloading engine that collects personal context of mobile users, learns predictive patterns of application usage, and preloads applications to improve their responsiveness. Compared with existing caching schemes such as Least Recently Used (LRU), Falcon improves the application responsiveness by 2 times.

We then study the problem of improving the accuracy of cloud-based image search by presenting a mobile service named CrowdSearch. In CrowdSearch, crowdsourced human validation is incorporated into the system to improve the search results from computational image search engine. Despite the fact that human validation is involved, CrowdSearch still provides realtime search response to mobile users with a delay of less than a few minutes. Compared with existing cloud-based image search, CrowdSearch achieves over 95% accuracy consistently across multiple categories of images, while the response time is kept within a few minute.

Lastly, we study the problem of energy- and bandwidth-efficient image search by presenting a mobile system named SenSearch. Unlike existing approaches that transmit raw images for search, SenSearch turns smartphones into micro image search engines, where images are collected, indexed, and searched via compact features that are two magnitudes smaller than a raw image. SenSearch improves the energy and bandwidth cost by five times compared with cloud-based image search engines where images are sent to a centralized server for search.

1.6 Thesis Outline

The rest of the thesis is organized as follows.

In Chapter 3, I present FALCON, a context-aware mobile system where user context is used for preloading applications, including prefetching network content, to improve the responsiveness of mobile apps.

In Chapter 4, I present CrowdSearch, a mobile image search service that combines cloud-based image search and crowdsourced human validation together to provide an accurate and realtime image search for mobile users.

In Chapter 5, I present SenSearch, an image search engine which exploits compact features to represent images, and turns smartphones into micro image search engines that collect images, index and search images by using the compact features.

In Chapter 6, I conclude the thesis and describe potential avenues of future research.

CHAPTER 2

BACKGROUND

This chapter presents background material on context-aware mobile computing and people-centric smartphone sensing. More detailed related work sections are also provided in the remaining chapters.

In §2.1, I describe the state-of-the-art context-aware mobile systems, and introduce the challenges of utilizing context for improving the performance of operating systems. In §2.2, I describe recent smartphone sensing systems and introduce the problem of image search for smartphones. We also provide a brief summary of crowdsourcing services that we exploit to improve the image search performance.

2.1 Context-aware mobile systems

This thesis explores the problem of how to leverage user context to improve the performance of mobile operating systems. Before presenting our contributions, we provide a detailed background of recent advances in context-aware mobile computing, and we focus on inference methods that can extract different types of context from sensing data. We don't introduce new context inference algorithms in this thesis, but instead we take advantage of such inference methods to obtain the user context, and exploit the context to improve the performance of mobile OS.

2.1.1 Inference User Context from Sensing Data

Table 2.1 summarizes four major categories of context that can be inferred from sensing data, namely logical locations, activities, ambient environments, and virtual reality.

Table 2.1. Summary of mobile context and their inference methods

Context	Major Sensors	Major Inference Methods
Logical Location	GPS, WiFi, Cellular, FM	reverse geocoding, clustering w/ multiple sensory data
Activities	Accelerometer, Gyroscope, GPS	GMM, entropy, community similarity
Ambient Environment	Ambient light, microphone	bandpass filters, unsupervised learning
Virtual Reality	Camera, GPS	face recognition, pose tracking

Logical locations: Localization has become a standard functionality in smartphones. GPS and radio-based localization methods, especially those exploiting cellular networks and WiFi, have become common for recent smartphones. In recent years, we have also seen other localization methods such as using FM [20] and Bluetooth [14] for indoor localization.

Many applications exploit the localization capability of smartphones to infer the logical location of mobile users. One example is reverse geocoding technique, which converts raw location coordinates into readable logical locations such as an address or name of an adjacent place. Major social network applications such as Facebook, Twitter and Foursquare exploit the reverse geocoding technique to allow users to share their logical locations with friends via "check-in" [32, 36, 110]. Many research have also used multiple sensory data from smartphoens to infer logical locations with high accuracy. For example, Cenceme [78] and SurroundSense [11] propose schemes to detect the logical location of mobile users with high accuracy by using a combination of sensory input from smartphones including GPS, WiFi, and Cellular information.

Activities: While logical location answers the information of "where are they", activity of mobile users answers the question of "what are they doing". Motion sensors such as accelerometer and gyroscope can be used to detect the activities of users including driving, walking, biking, sitting, running, and standing through various inference methods. For example, machine learning methods such as Gaussian Mixture Models (GMM) is used to classifier activities [37, 56, 60, 65, 66], frequency-domain entropy and correlation features are used for activity recognition [13, 21]. N. Lane et al introduces community similarity network to infer the activities of mobile users in large scale [55]. Many applications are built on top of the activities of mobile users, including recommendation systems such as [15], healthcare systems such as [108], and multimedia systems such as [29].

Ambient Environment Ambient environment defines the presence of surrounding environments for mobile users, such as people or other phones in the vicinity. Many sensors such as acoustic sensors like microphones, ambient light sensor, and compass, can be used to detect the surrounding environment [89]. The ambient environment in turn can be used to infer the activities or locations of mobile users [10, 43, 119]. For instance, Wang et al proposed a framework where audio recognition is implemented using bandpass filters to detect the activities of users [125]. SoundSense [64] also proposes inference algorithms that classifies user locations and social contacts through unsupervised ambient sound learning.

Virtual Reality Virtual reality is another category of context where an augmented view of the physical world is presented to users by exploiting the sensory data from smartphones. An important aspect of virtual reality takes advantage of various vision algorithms to process the scenes captured by smartphone cameras. For example, Twitter-based augmented reality apps and Takacs use facial recognition technique for augmented reality [9, 105], Wagner et al exploit pose tracking technique for extracting objects from mobile phone images [122]. Other sensory information can be used to improve the user experience in virtual reality as well. For example, Musolesi et al proposed a virtual avatars application which uses smartphones to infer daily activities of mobile users, and use them to reinforce the avatars in virtual world such as Second

Life [58]. Lifton et al proposed schemes to enhance the user experience of the virtual reality by using a combination of light, sound, motion, and temperature [59].

2.1.2 Challenges for Context-aware OS

Although smartphones can be used to sense a wide spectrum of user context including location, activities, and surrounding environment, how to exploit the user context for improving the performance of smartphone operating systems is still an unsolved problem. In this thesis, we explore the opportunities of using user context for better OS performance, and specifically we focus on the following challenges:

- How to find predictive patterns of user context and how to take advantage of the context pattern to improve the mobile operating system performance?
- How to design context-aware mobile OS such that the energy cost for acquiring user context and mining context patterns is bounded?

In §3, we are discussing how to solve these two problems by presenting a context-aware mobile OS where we use application preloading as a driving example.

2.2 Image Search for Smartphones

Besides context-aware mobile OS, another focus of this thesis is image search for smartphones. In this section, we provide background of image search for mobile phones, and we focus on the following aspects: first, we provide a brief summary of mobile systems that exploits the camera sensor of smartphones. We then narrow down to the image search application, and present a concise summary of the major techniques used to enable image search. Finally, we present the background of online crowdsourcing services and summarize its advantages and disadvantages compared with automated image processing techniques.

2.2.1 Image Search Application

Sensing using mobile phones has become popular in recent years. Examples include Urban sensing [17], Nokia's Sensor Planet [114], MetroSense [31], and many others. The emphasis of such efforts is to utilize humans with mobile phones for providing sensor data that can be used for applications ranging from traffic monitoring to community cleaning. Much recent work has looked at mobile sensing and people centric sensing from various perspectives, ranging from sensing platforms to inference of sensing data [11, 30, 64, 75, 76, 77, 78, 79, 82, 93].

Our thesis focus on one specific application — image search, where images are captured and used as a query to search through image search engines for relevant information, such as the name of the object in the query image. Goggle Goggles [41] is an example of image search engine, where certain categories of images, such as landmarks and textbooks, are indexed and can be searched by using phone-captured images as queries. A relevant mobile system that facilitates image search is the iScope system [133], which is a multi-modal image search system for mobile devices. iScope performs image search using a mixture of features as well as temporal or spatial information where available. While using multiple features can help image search engine performance, it does not solve the problem of low accuracy for certain categories.

Our vision includes innovations in two aspects. First, we aim at improving the performance of automated search by incorporating crowdsourcing-based human validation. Second, we extend the image search to smartphones where each mobile device is turned into a micro image search engine with high energy efficiency.

2.2.2 Image Search Techniques

To achieve the vision of the thesis, we exploits the recent technology advances in image processing and information retrieval. First, recent advances in object recognition makes it possible to efficiently compute and store compact image representations. For example, SIFT features of images represent the corner and contour change of objects, thereby describes the objects inside an image [61]. However, SIFT features has a disadvantage that their size is too large. One SIFT feature is a 128 dimensional vector, which makes a set of SIFT features can have the same size as the image they represent. In [33, 99], a new feature referred to as visual terms or visterms is introduced to compress the SIFT features to smaller size. Visterms have the size of two magnitudes smaller than SIFT, which makes it very compact for indexing, storage, and search. Second, recent advances on image search by using techniques adapted from text retrieval [86, 87] makes image search viable to smartphones. The image search process is built on top of a a large vocabulary of features such as visterms, and are generally referred to as search by “a bag of words” [23, 85, 87]. This image search pattern makes it feasible to index a large volume of images with fast query speed. It also makes it possible to enable image search on resource-constraint devices such as smartphones.

2.2.3 Online Crowdsourcing Services for Image Processing

We also exploit online crowdsourcing services such as Amazon Mechanical Turk (AMT) to improve the performance of image search engines for mobile users. In this section, we provide a brief overview of current research on online crowdsourcing services, and explain the advantages and challenges of exploiting crowdsourcing for image search.

There have been numerous crowdsourcing services that exploits a large number of online people to provide services that requires human intelligence. Luis Von Ahn’s pioneering work on reCaptcha [121] uses humans to solve difficult OCR tasks, thereby enabling digitization of old books and newspapers. His another work on the ESP game [120] uses humans for finding good labels for images, thereby facilitating image search. The two systems use different incentive models — reCaptcha protects websites against robots, whereas ESP rewards participants with points if the players provide matching labels. Other popular crowdsourcing models include the auction-based crowdsourcing (e.g. Taskcn [115]), and simultaneous crowdsourcing contests (e.g. TopCoder [116]). Our work is inspired by these approaches, but our goal is to solve the problem of using crowdsourcing to provide *real-time* image search services for mobile users.

Micro-payment crowdsourcing service is a new type of online crowdsourcing service, where users participate to simple tasks that usually take a few seconds to a few minutes to complete, and the incentives for such tasks are relatively in small amount, typically a few cents. The most common example of micro-payment crowdsourcing service is Amazon's Mechanical Turk (AMT), where tens of thousands of online users can be exploited to complete tasks that require human intelligence. Many applications have begun to utilize micro-payment crowdsourcing systems such as AMT to process images. This includes the use of crowdsourcing for labelling images and other complex data items [53, 96, 101]. For example, Sorokin et al. [101] show that using AMT for image annotation is a quick way to annotate large image databases. There have been several studies to understand quality of results from AMT solvers [53, 96].

Advantages of online crowdsourcing services for image search: Compared with automated image processing techniques, humans can interpret image contents with better accuracy. With online crowdsourcing services, it is feasible to exploit tens of thousands of online participants to process image contents, such as tagging images. Volunteer-based crowdsourcing such as Google Labeller and micro-payment crowdsourcing services such as AMT also provide feasibilities to process huge volume of images with low cost.

Challenges of exploiting crowdsourcing for image search: Although humans are good at distinguishing image contents, there are also a set of challenges to tackle.

First, humans have biases and errors, which can lead to inaccurate image processing results. For example, people may fail to distinguish if two images contain the same objects or not if the images are very similar or if the participants are not familiar with the subjects. Second, human processing is much slower than automated image processing. Validating the content of an image can take a few seconds to a few minutes by a human participant, and such slow processing speed is a hurdle to integrate the human processing into the image search process with realtime response. Third, human processing requires incentives. Although micro-payment crowdsourcing provides a feasibility of reducing the cost of crowdsourcing tasks to a few cents each, the monetary cost is still not negligible if large volume of images are processed by human participants. In this thesis, we focus on explore the tradeoff of the accuracy, latency, and incentive of crowdsourcing-based image search for mobile phones.

CHAPTER 3

FALCON: FAST APP PRELAUNCHING BY EXPLOITING MOBILE PHONE CONTEXT

3.1 Introduction

In this chapter, we study the problem of how to exploit mobile context to faciliate the design of mobile operating systems. In particular, we examine how mobile context can be used to boost up the loading speed of mobile applications to improve their responsiveness.

Mobile apps have blossomed in popularity and ubiquity, and have become the central usage of mobile phones. With each generation, the multitude and diversity of apps continues to grow. A recent tally revealed over 500,000 iOS apps, 350,000 Android apps and 100,000 Windows Phone (WP) apps, covering a large spectrum of categories such as games, productivity, finance, travel, personal infromation management, healthcare, and many others. With so many apps that can run on mobile phones, systems support that can improve our daily app interaction experience is poised to be widely beneficial.

Early pioneers in mobile computing recognized that during mobile interactions, human attention is a scarce resource [40, 95]. Our analysis of a long term user study [97] supports this view: 50% of mobile phone engagements last less than 30 seconds, and 90% of engagements last less than 4 minutes. With such brief periods of interaction, it is all the more important that interaction is rapid and responsive.

Unfortunately, interacting with apps can be surprisingly sluggish. Applications can take dozens of seconds to launch, negating any sense of agility. A particularly troubling class of apps are games, some of which can take upwards of 20 seconds to fully launch past splash screens, and even more time to reach a playable state. Our investigation shows that even relatively simple apps such as weather reporters can experience launch delays of 10 seconds or more.

Some applications, such as email clients, are optimized for quick launch within seconds. However, these applications mask actual latency by showing stale content while asynchronously fetching new content over the network. While appropriate for a restricted set of interactions, stale content fails to engage the user in the same way that the latest fresh content can. When considering the full launch time starting from touch until the latest content is displayed, apps with asynchronously loaded content can also exhibit severe load times of up to a dozen seconds.

One possible approach to decrease launch time is to cache apps in memory. This approach is implemented by iOS, Android and WP, yet suffers from several drawbacks. First, some apps, especially games, demand large amounts of memory. Keeping large applications such as games in the memory will lead to overwhelming memory real estate for other apps. Therefore, naïve caching schemes can exhibit low benefit. Second, caching does not address the issue that content may become stale by the time the user interacts with the app.

Another possible approach that addresses stale content is the use of push notifications. While push notifications can ensure content freshness, the energy cost of push communication can be prohibitively high [90]. Not surprisingly then, four out of the top five battery saving suggestions offered by Apple concern disabling push notifications [5].

Ideally, we would like to take advantage of both available cache and proactive content updates without incurring their drawbacks. In fact, by proactively preloading applications ahead of the time when they are actually used, the responsiveness of applications can be significantly improved and the network content can be fetched with fresh state. However, app preloading comes with a cost – incorrect preloading will incur both energy and memory cost. So the core question we want to tackle is how to preload mobile apps with high accuracy to improve the responsiveness of mobile apps while avoid high energy and memory cost.

It so happens that mobile devices are equipped with an increasingly sophisticated array of sensors such as accelerometers, cameras, microphones, and geolocation units. Each can provide insightful context for the OS [22], such as logical locaitons, motion status (such as driving, walking, and running), and many others.

We observed that the application usage has a tight correlation with mobile phone context. For a particular user, many applications, such as games like Angry Birds, are more likely to be used at home, whereas Email and Calendar usage may be much more likely at work. Or, a user may habitually check Facebook after receiving an SMS when at school. This observation motivates us to design a mobile operating system, where applications are preloaded according to the context.

In this thesis, we present the design and implementation of FALCON[1] system to predictively *prelaunch* apps based on context signals and user access patterns. Prelaunch entails both preloading apps in memory and app-specific fresh content preparation.

At the foundation of FALCON is a data-driven analysis that reveals surprising insights about how and when we use mobile apps. We find an intrinsic bundling behavior in how mobile apps are used — once a user starts using the phone, say for a phone call or SMS, they tend to use several other applications before turning away from the phone to another task. Bundling behavior manifests not just at the timescale of single user session with the phone, but across longer durations of several weeks. We find that games are often used in bursts where users tend to obsessively

[1] Fast App Launching with Context. The peregrine falcon can swoop dive at more than 200 mph – making it the fastest animal on earth.

use a game for a few weeks, and then ignore it entirely for several months. Similarly, other contexts such as location and time-of-day plays a central role in app usage, perhaps indicative of our tendency toward structured behavior.

While such behavioral insights can be distilled into a set of preload prediction rules, a key challenge is that users are not alike in how these behaviors apply to them. Some users are more likely to open several apps after a phone call whereas others do so after opening the browser; some users have strong tendencies at a core set of tightly defined locations such as home and work whereas others exhibit more diversity. To address this, FALCON personalizes the context triggers to the individual to maximize prelaunch accuracy.

Applications also differ widely in terms of the benefits from correct prelaunch versus the cost of incorrect prelaunch. As an example comparing two popular games, *Need for Speed* takes over 2× more energy than *Plants vs. Zombies* even though both load equally slowly. FALCON uses a novel cost-benefit adaptive prediction algorithm that accounts for both the expected latency benefit of correct predictions and the expected energy cost incurred for wrong predictions. The algorithm adapts to budget constraints and jointly optimizes the applications to prelaunch as well as the temporal and location contexts under which such prelaunching is enabled. Both features and learning algorithm are designed to achieve very low runtime overhead.

We have implemented FALCON as an operating system modification to the Windows Phone 7.5 operating system and show that it has minimal energy, memory and processing overhead. In addition to preloading apps from local storage, FALCON ensures content freshness by prefetching content before app launch. This is achieved with a WP event handler that FALCON invokes whenever it predicts the app is about to be launched. For example, a Twitter app can fetch new tweets during this event. As a result, a user's time to first meaningful app interaction is no longer bound by network latency.

Our major contributions are as follows.

- From extensive data analysis, we design spatial and temporal features that are highly indicative of mobile app access patterns.
- We design a cost-benefit learning algorithm whose predictions lead to launch latency reduction of nearly 6 seconds per app usage with the daily energy cost of no more than 2% battery lifetime. Predictions also lead to getting content that is only 3 minutes old at launch without waiting for content to update at launch.
- We prototype FALCON on a Windows Phone, and demonstrate that it has minimal runtime energy overhead of a few μAh and a small memory footprint of under 2 MB.

Taken as a whole, FALCON effectively replaces a subset of the policy decisions traditionally reserved for the memory manager and scheduler. Context permits FALCON to make much more informed app decisions which lead to substantial launch time reductions at minimal overhead.

The rest of this chapter is organized as follows. §3.3 provides through literature rewiew and quantitative background of the app preloading problem. §3.4 provides an overview of the system. §3.5 discusses the launch predictor, features and learner. §3.6 details implementation aspects. §3.7 evaluates FALCON. §3.2 discusses related work, and §3.8 offers a discussion and conclusion.

3.2 Related Work

3.2.1 Existing Application Launching Optimization Schemes

Systems optimizations for faster application launching has a long history on traditional computational systems. One area that has received significant treatment is the arrangement of code and data on disk for faster loading. Windows sorts code and data blocks on disk to minimize disk head movement during launch, and defragments files on disk for faster launching. Intel TurboMemory supports pinning selected applications' blocks to high performance cache to speed launch times. Joo et al. [50] investigate interleaving I/O fetch and code execution to increase app launch times. These schemes are complementary to our use of context for making prelaunch decisions.

SuperFetch, a part of Windows OS, is an extension to the memory manager that tries to predict future code and data accesses and preload them. The scheme works by learning the user's favorite apps based on time of day and day of the week, and loading these when there are free pages available. Unlike FALCON, SuperFetch does not use predictive mobile context such as location nor energy budgets in its loading decisions. Whereas both FALCON and SuperFetch interface with the memory manager to load apps, FALCON further reduces the total launch time by calling an app's prelaunch routine.

3.2.2 Using Context for Optimizing Mobile System Performance

Recently, the authors of [22] proposed the use of user context information from sensors to improve mobile operating system performance, and suggested that OS-managed context could benefit user privacy, energy, system performance. The authors speculated that core system services – among others, memory management and process scheduling – could be beneficiaries of sensor context signals.

Browsers and other web systems have long known how to employ content prefetching and caching to decrease page load times [2]. A draft HTML specification includes `prefetch` attributes as hints to the browser [81]. These prefetch mechanisms generally do not guide prefetch decision making, as our launch predictor does, and are not cognizant of mobile context nor energy concerns.

Recently in [68], the authors studied mobile web usage data and built a predictive page loading mechanism based on dividing pages into regularly visited sites and spontaneously visited sites.

Several platform-specific tweaking programs allow mobile users to configure app loading according to preset rules [107]. The shortcomings of these programs are that they stop short of addressing total launch time, and manually maintaining a rule-based system for the typical user is challenging.

3.3 The Responsiveness Problem of Mobile Applications

Before presenting our context-aware mobile operating system where applications are prelaunched based on context, in this section, let us first take a closer look at the issue of the launch latency of mobile apps.

Our investigation is based on trace data collected from the Rice LiveLab user study [97], and our own internal study. In the Rice LiveLab's study, thirty four volunteers were given iPhones for a period of fourteen months from 2010-2011. During this time, their device usage was monitored. Our own internal double-blind study consisted of three WP users monitored over one month. We focus on the following elements which were recorded in both traces: *app usage, location, datetime.* The LiveLab's trace is much more extensive; it will be the main focus of investigation unless otherwise noted.

We first examine the thesis that mobile interactions are inherently brief [40]. Figure 3.1 illustrates the CDF of app usage durations across all users in the Rice trace. Indeed, most interactions with the phone are short with 80% of the apps being used for less than two minutes. This indicates that short engagements are the norm and the proportional effect of slow app startup can be expected to be even more severe for mobile users.

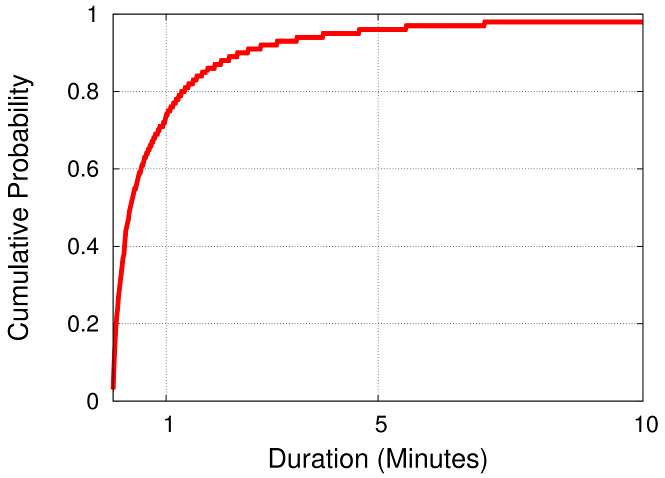


Figure 3.1. App usage durations for a representative individual over a one year long trace.

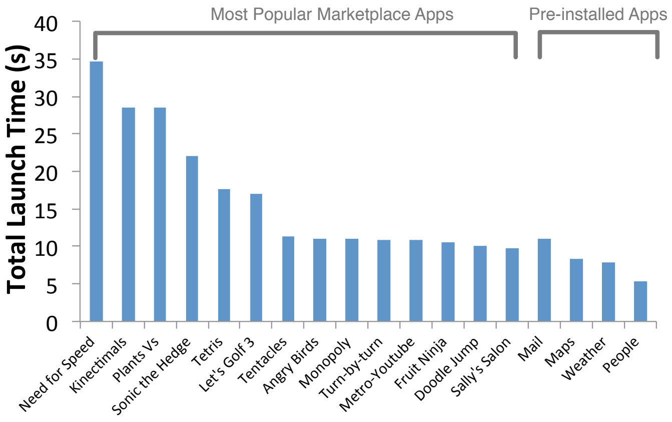


Figure 3.2. Total launch times of popular apps on Windows Phone. The first 14 are most popular apps in the Marketplace, and the last four are first-party apps that are preinstalled.

Qualitative reports suggest that slow app launch is a substantial drag on user experience. As an example, a recent iOS upgrade caused an outcry from iPhone owners that were hit with slower speeds [8]. Popular blogs suggest that users are willing to try involved system hacks to improve app launch times by ten seconds [7].

To quantify the extent of the issue, we studied the top 20 popular apps in the WP marketplace. *Launch time* represents the elapsed time from when a user taps on an app's icon to the first time a user is able to interact with the application. Figure 3.2 shows that launch times are fairly long, with a median launch time of 11.2 seconds and an average launch time of 16.7 seconds. Surprisingly, no app launched in under 5 seconds. The trend for iPhone was similar.

The use of asynchronous launch operations is a popular coping mechanism for slow launch times. Two typical classes of asynchronous operations are fetching content from the network while displaying stale content, and loading content from local storage while showing a splash screen. For example, when the native WP Email app is launched, it displays old messages while asynchronously fetching new messages. As a result, the launch time is 1.3-1.7 seconds. However, when the time for asynchronous message fetch is also accounted for, the elapsed time is 11 seconds, even on WiFi with no new emails present. Our anecdotal evidence suggests that checking for new email quickly is much more valuable than reviewing old email. Similarly, checking social networking apps for the latests posts is much more valuable than rereading old posts. For these network-bound apps, fast completion of asynchronous launch operations is important for much of the app's utility. Similarly, games commonly employ splash screens while asynchronously loading graphics and art objects into memory from local storage. To count total launch time in these cases, we disable or skip any splash screens. Therefore, in this work we focus on *Total Launch Time (TLT)* which we define as the base launch time plus completion of asynchronous launch operations. Figure 3.2 also shows the TLT of several first-party apps: Email, People (a Facebook and Twitter news reader), Maps and Weather. The TLT range is 5-11 seconds.

Slow launch times paired with brief usage durations are a troubling combination. It suggests that users are experiencing far too much overhead waiting for their apps to launch rather than engaging their apps. Fortunately, as §3.5 will show, our data-driven exploration indicates that user behavior is highly habitual, and linkable to observable context such as time and location.

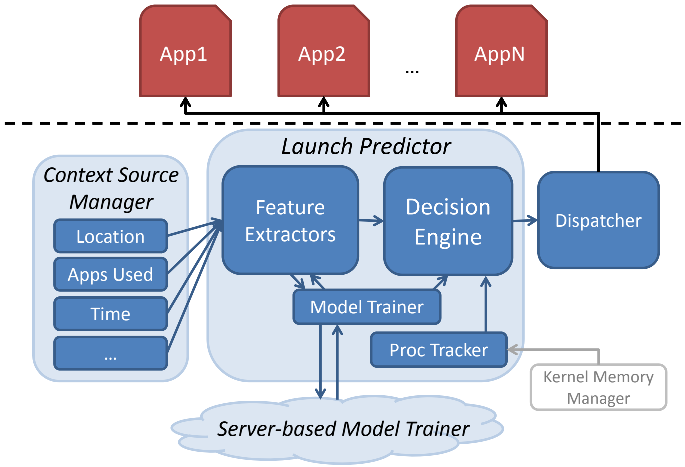


Figure 3.3. The FALCON System Architecture.

3.4 System Overview

FALCON offers an architecture in which to systematically observe and utilize context and execute predictive app launch actions. Figure 3.3 shows the FALCON system architecture. The central component is the launch predictor. Its role is to use context signals to predict app launches. The launch predictor's feature extractors convert raw data from context sources into features for the decision engine and model trainer. The decision engine performs *inference* to determine which features to use and what applications to prelaunch. The prediction is then passed on to the dispatcher, which loads apps into memory and executes the prelaunch routine of the selected app(s). The model trainer records observed features in order to periodically *train* and update the decision engine and feature extractors with new parametrizations.[2] Note that training occurs infrequently and is only critical when access patterns change significantly, whereas inference occurs frequently and is required whenever a launch prediction is made. The process tracker handles communication with the kernel's

[2] Unlike standard features, ours involve some training.

memory manager in order to ensure that the decision engine's view of the apps running on the system is in sync with what is actually running.

Supporting the launch predictor, the context source manager is a lifecycle manager and container for various context sources. It also helps shepherd raw data from individual context sources to the launch predictor.

A benefit of the FALCON architecture is its modularity. We have been able to use several context sources (location, time, ambient light level, accelerometer and gyroscope), and three separate decision engines within the architecture. In the following sections of this chapter, we focus on the instantiation of the FALCON architecture which we found most suitable for launch prediction.

3.5 Launch Predictor Design

The Launch Predictor is a machine learner with several feature extractors and a decision engine. This section first discusses the important features we identified, and then describes a novel cost-benefit learner and decision engine well-suited to launch prediction.

3.5.1 Personalized Features

Our primary objective for feature design is to identify a set of features that give strong insight into the next app to launch and that are very inexpensive to compute at inference time. As is typical of feature design, we take a data-driven approach. This investigation is heavily guided by our data exploration of the LiveLab and internal user studies. For each feature, we present a motivating vignette uncovered from the traces, followed by a formal definition of the feature. Table 3.1 summarizes the features considered.

feature	description
f_t	triggers
f_{dt}	dynamic triggers
f_λ	location cluster
$f_{dt,\lambda}$	dynamic triggers per location
f_{ToD}	time of day
f_β	burst behavior indicator

Table 3.1. Feature Summary

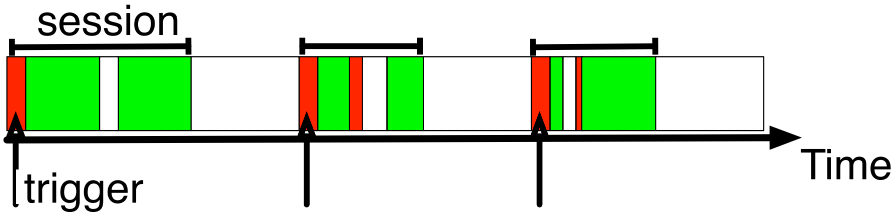


Figure 3.4. During this 2 hour timeline, SMS (in red) triggers other follower apps (in green). Arrows indicate start of triggered sessions.

3.5.1.1 Triggers and Followers

A novel feature in FALCON is session triggers and followers. Our data analysis reveals that once the user starts using the phone – for example to answer a phone call or view an SMS message – they have a tendency to open a sequence of other applications within relatively short durations of each other. We refer to such a sequence of application launches as a *session*.

The data traces reveal that certain apps are more likely than others to be the *trigger apps* that start sessions, whereas other apps are more likely to be *follower apps* that are used in the same session as the trigger apps. Figure 3.4 shows a two hour extract of one user's trace. During this time SMS was the trigger for four separate sessions. Analysis of aggregate user behavior revealed that SMS, Phone, Email, Facebook, and Web Browser are among the most popular triggers across all users. This is intuitive since trigger apps are ones that grab the users' attention (e.g. SMS and Phone) or satisfy immediate information needs (e.g. Browser).

Let f_t denote a trigger feature. Intuitively, the trigger features identify the occurrence of a trigger app weighted with the likelihood of a follower app. For random variable follower app F, trigger app T, and candidate launch app α, the trigger feature is calculated as follows.

$$f_t = \Pr(F = \alpha \mid T \in \tau_t)\,\mathrm{I}(\tau_t)$$

where τ_t is a set of triggers, for example $\{SMS, Phone\}$, and $\mathrm{I}(x)$ is an indicator function that is 1 if (any member of) x is observed, and 0 otherwise. The term $\mathrm{Pr(F = \alpha \mid T \in \tau_{st})}$ expresses the probability that the follower app is the candidate launch app α given that the trigger app is one of SMS or Phone. Note that probabilities are based on the data distribution observed during training.

Interestingly, we found that the best triggers were different for different applications, even for a single individual. For example, the best three triggers for Email are different from those for AngryBirds. There is also significant variability across users — a heavy user of games might have a game as a trigger whereas others may not. These differences lead us to propose dynamic triggers that are calculated on a per-launch-candidate basis as the set of top-k triggers most likely to lead to the launch candidate as a follower.

$$\hat{\tau}_k = \operatorname*{argmax}_{\tau_k} \Pr(F = \alpha \mid T \in \tau_k)$$
$$f_{dt,k} = \Pr(F = \alpha \mid T \in \hat{\tau}_k)\,\mathrm{I}(\hat{\tau}_k)$$

where $\hat{\tau}_k$ represents the k best triggers for α. §3.5.2 discusses the procedure for choosing k to arrive at a fixed set of dynamic triggers per app f_{dt}.

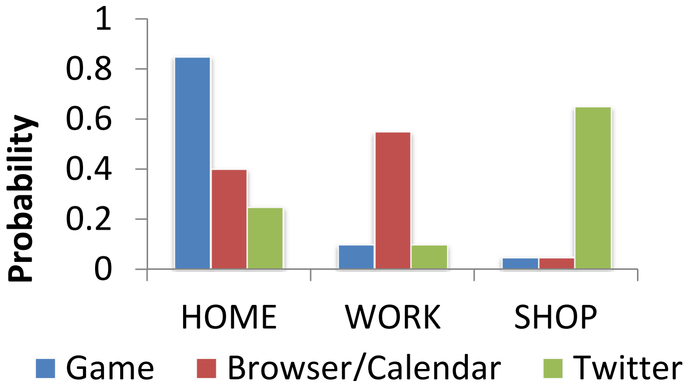


Figure 3.5. App usage is correlated with location.

3.5.1.2 Location Clustering

Empirical observations suggest that location is correlated with app usage. Figure 3.5 illustrates the effect of location on app usage for a brief two week time window for a representative user from our internal user study. For this user, there is a surprisingly strong tendency for game usage at home, Browser and Calendar usage at work, and Twitter usage at a frequently visited shopping center.[3]

The location feature f_λ is computed by assigning the user's current location to the nearest neighbor cluster.

$$f_\lambda = \underset{c \in Clusters}{\text{argmin}} \ \text{DIST}(\lambda, l_c)$$

where λ is the current location and l_c is the central location of cluster c. Clusters are computed from the user's historical data during the training phase. The literature offers many geospatial clustering algorithms [74], and after experimenting with several, we chose k-means clustering which incurs low computation cost while performing comparably to the others.

[3] The labels Home, Office and Shop are educated guesses according to the recorded geocoordinates.

We can use location in conjunction with triggers as a joint feature $f_{dt,\lambda}$. For each location, we compute the best set of dynamic triggers as follows.

$$\hat{\tau}_{\lambda,k} = \underset{\tau_{\lambda,k}}{\operatorname{argmax}} \Pr(F = \alpha \mid T \in \tau_{\lambda,k}, L = \lambda)$$
$$f_{dt,\lambda,k} = \Pr(F = \alpha \mid T \in \hat{\tau}_{\lambda,k}, L = \lambda)\, \mathrm{I}(\hat{\tau}_{\lambda,k})\, \mathrm{I}(\lambda)$$

This feature captures dynamic trigger variation across locations.

One caveat with location clustering is that while some clusters are well defined with dense samples, many location samples are sparsely spread out across wide areas. The user has typically only visited these locations once or rarely, and therefore do not appear to be part of any well-defined cluster. Therefore, an alternate approach is necessary for dealing with sparse locations.

We introduce the logical location feature $f_{\lambda\lambda}$ to address the issue of sparse locations. The basic idea is to treat all coffee shops similarly, whether it is the user's frequently-visited coffee shop, or it is a new coffee shop that the user is visiting for the first time. The procedure works as follows. After location clusters are computed according to the above procedure, each dense cluster is assigned a logical location from a business directory's taxonomy. The assignment is made automatically based on reverse geocoding the cluster center. We currently employ the YellowPages directory [28] for geocoding and taxonomy assignment. Example taxonomy categories include: "Sports & Recreation", "Government & Community" and "Food & Dining." Upon encountering a sparse location, it is reverse geocoded and assigned a logical location. It is then treated as if it were a part of a dense cluster with the same logical location.

3.5.1.3 Temporal Bursts

We also find that a user's app usage changes significantly over longer time scales. Figure 3.6 illustrates an app, Angry Birds, becoming popular during a month-long

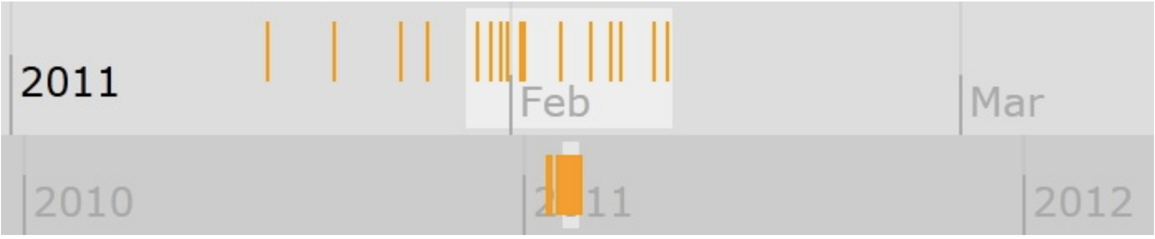


Figure 3.6. This particular user launched the app Angry Birds frequently from mid-Jan to mid-Feb only. Both yearly and monthly timelines are shown.

period, then ceasing to be used at all afterward. Perhaps not surprisingly, this behavior turns out to be very common for games and entertainment apps. However, our data analysis reveals that such *burst* usage behavior is dependent on the type of application, for example, games have shorter but more intense bursts in comparison with other applications. Therefore, a key challenge is to determine the frequency and duration of these bursts for each application in an efficient manner.

To address this issue, we develop a novel *burst predictor*. The burst predictor uses multiple sliding windows, each of a different resolution, to detect and react to sudden upticks in app usage. Each app has its own burst predictor to track its bursty behavior.

As a basis for our burst predictor, we use a wavelet-based burst detection algorithm [134] as a subprocedure. Given a stream of data, the detector's goal is to find all windows of any length during which the aggregate statistic of the window exceeds a threshold. In our instantiation, this corresponds to detecting any sequence of days during which an app is launched more than a threshold per day on average. We currently set the threshold to one.

The wavelet detector uses a unique modified wavelet data structure in order to bound the computational complexity of detection by $O(n)$ where n is the number of days, whereas the brute force solution is $O(n^2)$. The output is an indicator for each window $w_{i,j}$ where i indicates the offset from the start of the stream and j indicates the length of the window. Let $b_{i,j}$ be a burst indicator which corresponds to a sliding window starting from day i of length j days. $b_{i,j}$ is true if the corresponding sliding window is a burst, and false otherwise.

Provided the burst detector, our burst predictor must decide whether the next time step is part of a burst or not. To make the decision, our predictor measures the predictive power of each burst window size j in being able to successfully predict whether the app is used in the day following the window. Let w_j represent the predictive power of window size j.

$$w_j = \frac{1}{n-j} \sum_{i}^{n} \left[b_{i,j} b_{i+j,1} + (1 - b_{i,j})(1 - b_{i+j,1}) \right]$$

By analyzing historical data, we can now determine the best window size as the one with the highest score, and use it to detect a burst. This window size can be used to compute f_b, which is set to 1 if we are currently within a burst window and 0 otherwise.

$$\hat{j} = \underset{j}{\operatorname{argmax}}(w_j)$$

$$f_\beta = b_{n-\hat{j},\hat{j}}$$

From empirical observation, an algorithm modification is possible without affecting the overall algorithm performance. It is not necessary to consider window sizes longer than thirty days, since burst behavior is not observed on this longer timescale.

Time of Day Feature: Besides multi-day temporal bursts, app usage likelihood changes throughout the day as well. Figure 3.7 illustrates this for a typical user in our internal user study. A simple Time of Day (ToD) feature f_{ToD} represents the time

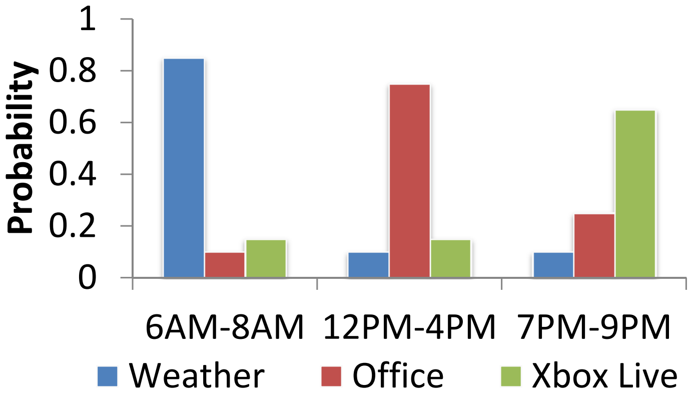


Figure 3.7. Time of day is correlated with app usage.

of the day, segmented by hour. Our experiments with finer and coarser granularities did not yield much difference.

3.5.1.4 Additional Context Signals

We also built context sources mapping to accelerometer, gyroscope and ambient light level sensors. These proved sufficiently straightforward to incorporate into the FALCON architecture. However, we do not yet have sufficient user trace data to analyze their effectiveness at predicting app launches.

3.5.1.5 Multi-feature Decision Engine

The different features that we have described can be used as the core inputs of a decision engine that makes application prelaunch decisions. Once training phase has provided the trigger features location clusters, and burst predictions, the decision engine can decide whether or not to launch an application α when a trigger T is met. The decision procedure takes three steps.

1. f_β is checked to see whether a burst of α is underway.

2. f_λ is checked to see whether the current location suggests use of α.

3. $f_{dt,\lambda}$ is checked to see whether a trigger for α started the current session.

The three steps are executed in a sequential order. The launch decision for α can be summarized as follows.

$$decision(\alpha) = \begin{cases} true & \textbf{if} \quad f_\beta \; f_\lambda \; f_{dt,\lambda} > 0 \\ false & \textbf{else} \end{cases}$$

Our description of the decision engine, however, has a major omission — it does not take into account the actual cost and benefit of a correct preload versus an incorrect preload. In the following section, we describe how we can incorporate these cost-benefit considerations into our decision engine.

3.5.2 Cost-Benefit Learner

The Cost-Benefit Learner (CBL) combines the output of the decision engine for each application together with knowledge of the cost-benefit tradeoffs of launching the application to decide whether to preload an application. We first provide an intuition for how the CBL operates before launching into the specifics of its mechanics and the details of the cost-benefit trade-off.

3.5.2.1 CBL Overview

To decide whether to prelaunch or not, CBL takes three steps: First, it computes the precision and recall metrics for an application from the decision engine (§3.5.1.5). Second, it converts the precision and recall number to cost and benefit. Third, it determines the best followers for each trigger by maximizing benefit for a fixed cost budget.

Precision and Recall: The precision and recall for a trigger and follower pair $\{T, F\}$ at location cluster λ are defined from observed user data as follows.

$$precision(T, F, \lambda) = \frac{\text{launches of } F \text{ after } T \text{ at } \lambda}{\text{sessions with trigger } T \text{ at } \lambda}$$
$$recall(T, F, \lambda) = \frac{\text{launches of } F \text{ after } T \text{ at } \lambda}{\text{sessions with follower } F \text{ at } \lambda}$$

Cost and Benefit: We measured the energy cost and loading time of all apps apriori. We can use the measurement results to determine the cost and benefit of prelaunching by combining the precision and recall results.

Precision describes the expected number of prelaunchs that leads to an actual usage. Multiplying by energy cost of a prelaunch, we can derive the energy cost of a prelaunch that leads to an actual usage. Recall describes the probability that a follower is preloaded. Multiplying by the launch time of the follower, we can derive the loading time benefit of a prelaunch across all launches. We will quantitatively analyze the cost and benefit in the next section.

Trading-off cost and benefit: We assume that the CBL is provided a daily energy budget. The energy budget is a predefined percentage of the total battery capacity that the CBL engine can use for its prelaunches within a day. (For example, the CBL may only be able to use 1% of the battery each day across all prelaunches.) Given this constraint, the goal of the CBL is to pick, for a given user, the best set of followers to prelaunch. Intuitively, we should prelaunch the items that have least energy cost and highest loading time benefit. The goal of our optimization framework is to choose the best option given the energy budget.

Intuitively, a conservative CBL would choose applications with higher precision and therefore fewer wasted pre-launches (i.e. less wasted energy), but it would also likely lead to lower recall. On the other hand, an aggressive CBL would pre-launch often and lead to high recall but may incur more wasted energy. The goal of our optimization framework is to choose the best option given the energy budget.

3.5.2.2 CBL Optimization Framework

We first describe the optimization procedure which runs during training. For ease of exposition, we cover the CBL's method for finding dynamic triggers on a per location- and per app-basis. In our implementation, the technique is extended in the straightforward way to additionally include time-of-day.

Let c_α and b_α represent the energy cost and latency reduction benefit respectively for a single prelaunch of α. Then the expected cost c' and benefit b' of selecting to prelaunch α for all encounters of t and λ is calculated as follows.

$$b'_{t,\lambda,\alpha} = b_\alpha \times recall(t, \alpha, \lambda)$$
$$c'_{t,\lambda,\alpha} = c_\alpha \times (1 - precision(t, \alpha, \lambda))$$

Note that while the benefit b' is scaled by the proportion of triggers leading to the follower, the cost c' is scaled by the proportion of triggers *not* leading to the follower. This is because mistaken predictions incur a cost c_α whereas energy for correct predictions can be accounted toward the user's actual launch.

We map our formulation to the 0-1 Knapsack problem to model the cost and benefit trade-off. Consider all tuples $< t, \lambda, \alpha >$ as candidates for the Knapsack. The predefined energy budget corresponds to the max allowable Knapsack weight, the expected reduced launch time $b'_{t,\lambda,\alpha}$ corresponds to the benefit of inclusion in the Knapsack, and the expected energy cost $c'_{t,\lambda,\alpha}$ corresponds to the weight in the Knapsack.

It is a 0-1 Knapsack since a tuple can be included or excluded in its entirety. The 0-1 Knapsack model for the cost-benefit trade-off is formulated as follows.

$$\text{maximize} \sum_{t,\lambda,\alpha} b'_{t,\lambda,\alpha}\, \mathrm{I}(<t,\lambda,\alpha>)$$

$$\text{subject to} \sum_{t,\lambda,\alpha} c'_{t,\lambda,\alpha}\, \mathrm{I}(<t,\lambda,\alpha>) \leq W$$

As before, the function I() is either zero or one, indicating whether or not to include the tuple $<t, \lambda, \alpha>$. W is the energy budget.

One possibility to solve the Knapsack problem is with dynamic programming. However, a dynamic programming solution requires pseudo-polynomial time. Instead, we use a standard greedy approximation to solve the Knapsack problem. The greedy approximation sorts the tuples in decreasing order of their benefit per unit of cost. It then proceeds to pick the tuples according to the ranked order, starting with the top-ranked one, until the energy budget is reached. The output of the greedy approximation is K, a list of tuples sorted by benefit per unit cost within the energy budget.

The final step of the optimization updates the dynamic trigger feature $f_{dt,\lambda}$ to use the appropriate top-k triggers. Recall from §3.5.1.1 that for a given α, the value of k is used by $\hat{\tau}_{\lambda,k}$ for the feature $f_{dt,\lambda,k}$. k is set according to the number of appearances of α in K. The decision engine, which looks at $f_{dt,\lambda,k}$ for the chosen k, performs inference in time linear to the length of K.

Dynamic Energy Constraints The Knapsack optimization works well if the user's behavior during each day[4] is representative of the user's aggregate behavior. However, there are times when the user may use more (or less) apps at the beginning (or end) of the day. We address this issue by

There are several ways to define an energy budget for the Knapsack optimization framework. For example, one could budget a fixed percentage of the battery per day, pick a percentage of the current battery level or have an finer-grained budget at an

[4] As a simplifying assumption, we take one day to correspond to one charge cycle.

hourly level. While our optimization can work with different definitions of the energy budget, we use the representative example of a budget that is chosen based on the residual energy.

To account for daily variations in app launch behavior and battery drain, we let Knapsack support an energy budget that varies over time. Rather than setting the energy budget statically, we set it based on the residual energy. The Knapsack model is adapted to the dynamic energy constraint W' as follows.

$$W' = (W - U)^{\theta}$$

where U is the energy consumed so far by prelaunching, and θ is a tuning parameter in the range $(0, 1]$ that can be chosen to discourage aggressive prelaunching when energy is abundant. We currently set $\theta = 1$. Note that meeting the dynamic budget constraint does not require recomputation of Knapsack. Instead, we maintain a pointer to the position in K which represents the dynamic energy budget cutoff.

Eviction Policy: Most phone OSs use an LRU policy for app eviction.[5] While the CBL is primarily used for launch prediction, the sorted tuple list K is also used to evict the app least likely to be used. As part of launch prediction, apps already resident in memory are assigned very low cost scores. Nonetheless, such an app may still be evicted due to its meager benefit b' according to the current trigger t and location λ. This conveniently integrates launch prediction with eviction policies.

For each app eviction candidate α, K is searched for an entry $< t, \lambda, \alpha >$ where t is the current trigger and λ is the current location. The eviction candidate whose entry is the lowest on the list is evicted. Since K contains only the optimal Knapsack

[5] Without paging, apps are evicted in their entirety.

tuples, in the case that an entry for an eviction candidate cannot be found in K, we revert to LRU.

Server-based Training If the user app launch behavior changes (e.g. a new location is visited or a new app is installed), retraining is recommended. Even though the greedy Knapsack approximation is efficient, training involves scanning through all the logged context data. Therefore, we offload training to a server. We currently initiate retraining manually. The log with data records since the last training are sent to the server, and new parameters are sent back to the phone.

Memory Constraints In the preceding discussion, we treated all preloads equally in terms of memory cost. The FALCON implementation and its associated evaluation also make this simplifying assumption. In practice, apps vary in memory usage. A logical extension of our Knapsack model can handle multiple-constraints where both energy and memory are treated as constraints.

3.6 Implementation

We have implemented FALCON as an OS modification to Windows Phone (WP) 7.5. We first discuss the elements of WP's app model relevant to our implementation before describing the implementation details.

3.6.1 Background on Windows Phone Apps

WP apps are similar to webapps in organization. Each app is collection of pages with links pointing across pages. An app is referenced by its unique ID (UID), and a page within an app is referenced by its page name. Pages can take arguments as parameters. The following is an example of these elements: *app://¡app-id¿/ShowContactInfo?id=8*. Each app has a default page *Default* that is loaded at startup. Apps can also support multiple entry points by supporting "deep links" to specific pages. For example, the standard Address Book app supports pinning of contact cards to the home screen via deep linking. Links are not functional cross-app; WP enforces strict app isolation.

The WP app lifecycle consists of four states: Not Launched, Active, Deactivated and Tombstoned. Apps transition from Not Launched to Active when tapped from the home screen, and from Active to Deactivated whenthe user navigates away to the home screen or another app. An app undergoing deactivation has a 10 second grace period to perform cleanup activity before receiving no further cycles. A deactivated app is still resident in memory until it is evicted by the memory manager which then changes its state to Tombstoned. For Tombstoned apps, the OS retains some minimal tombstones of bookkeeping information, such as the last page of the app the user was on. In very low memory conditions, even tombstones may be dropped. To limit battery drain, background apps are only permitted 15 seconds of CPU time every 30 minutes.

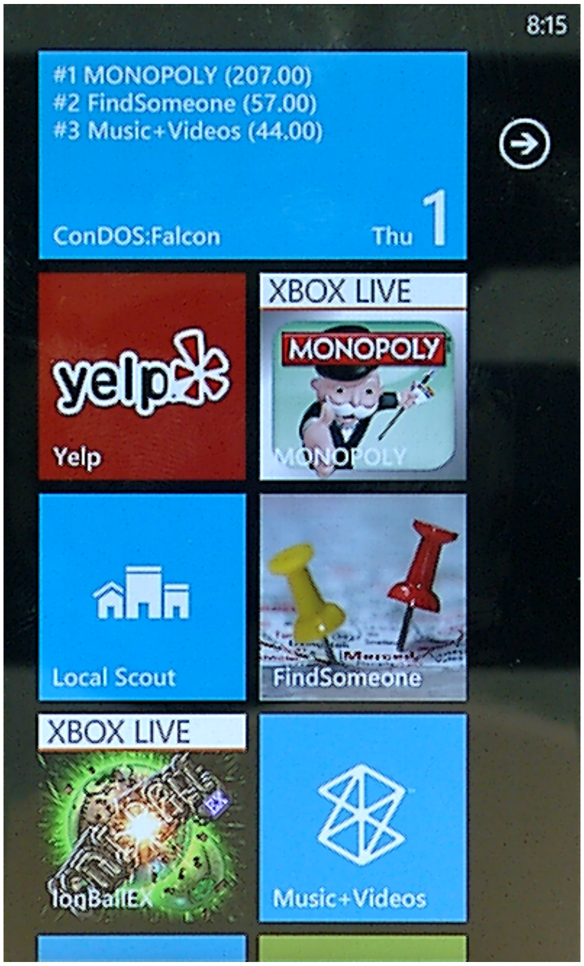
3.6.2 Implementation Description

FALCON is implemented in three parts. The first is an OS service. The second is a new application event upcall. The third is an ambient display widget.

The FALCON OS services is a privileged daemon written in native code. As such, it has access to the registry, a persistent structured datastore, and can call privileged library and system calls. For example, starting a new process is a privileged call. FALCON replaces a subset of the standard policies of the system's memory manager and scheduler with its prelaunch decisions.

An alternative FALCON implementation might have been to directly modify the system memory manager and scheduler. We chose a privileged service implementation because it is more practically deployable.

The service manages a thread for each context source. Currently, three context sources are used corresponding to the features described in §3.5.1. Context sources



#1 MONOPOLY (207.00)
#2 FindSomeone (57.00)
#3 Music+Videos (44.00)

Figure 3.8. FALCON ambient display widget is shown at the top of the home screen. Current predictions are *Monopoly*, *FindSomeone* and *Music+Video*.

post updates at different rates, depending upon the rate of change of the underlying signal. In order to support variable update rates, the main service shares a *model* data structure with each source and the launch predictor. Each source independently updates the model with observations for the context for which it is responsible.

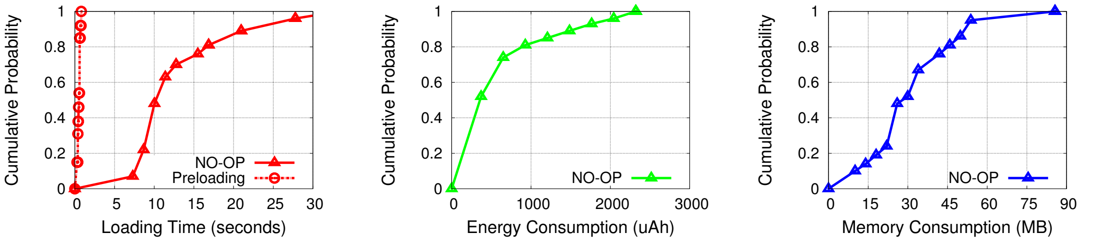
The launch predictor uses the model's observations in its inference process. The launch predictor is executed whenever one of several defined events occurs: (1) the phone receives an SMS or phone call, (2) the user unlocks the phone, or (3) the user launches a trigger app. These events already activate the phone's application processor from a low to high energy state so it is inexpensive to piggyback the launch predictor execution.

Upon generating an inference, the launch predictor passes the candidate app(s) to the *dispatcher*, which vectors the launch signal through the appropriate prelaunch event upcall. For third party apps which are written in managed C#, this process involves translating native calls to managed code calls.

Apps implement the event handler as a custom page named *Prelaunch*, e.g. *app://¡app-id¿/Prelaunch*. The dispatcher jumps directly to the Prelaunch deep link. An app can implement any custom prelaunch logic in its Prelaunch page. For example, our modified Email app fetches new emails during prelaunch, and our modified game loads the last level the user played during prelaunch. In general, network content fetch or local storage load operations are good candidates for Prelaunch. For apps that do not implement the Prelaunch link, the dispatcher opens its Default page instead. This is sufficient for us to reduce launch time for many apps without requiring app modification.

In WP, pages are meant to show a display to the user, which is not appropriate for prelaunching. Once the dispatcher opens an app to either the Prelaunch or Default page, it immediately deactivates it. With the existing WP deactivation grace period, this gives the app an opportunity to complete its app-specific prelaunch operation. For prelaunch deactivation, the grace period is modified to 25 seconds, which is high enough to cover the total app launch times of most apps. The deactivation event also gives apps an opportunity to adjust for any side-effects (e.g. unregistering server state) that the Default page may have initiated. Conveniently, developers already implement the deactivation event as part of the regular app lifecycle management protocol.

Outside of the dispatcher's actions, the system memory manager may independently load and evict applications. This can happen for example based on app



(a) CDF of launch time

(b) CDF of energy consumption

(c) CDF of memory consumption

Figure 3.9. Microbenchmark of launch time, energy cost, and memory cost. Apps measured include the top 20 most popular ones in Windows Phone Marketplace, and preinstalled first party apps.

launches that FALCON does not predict, or based on apps requesting more memory after launch that causes eviction of other apps. To ensure that the cost-benefit trade-offs of the CBL are accurately modeled, it is necessary to stay informed of the memory manager's actions. The *process tracker* monitors the current set of apps in memory and relays any changes to the learner. In order to implement CBL's cost-benefit eviction decisions, we rely on terminating applications. This also means that when we are interested in keeping a specific subset of apps in memory, we first kill any other apps.

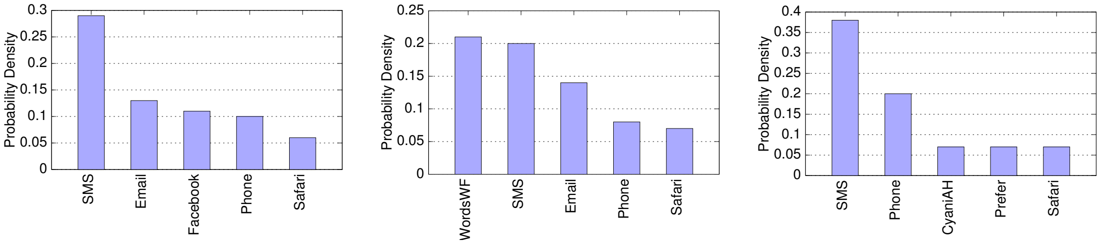
Lastly, FALCON includes an ambient display widget using the WP live tiles mechanism. Live tiles are simply home screen icons that can be updated programmatically. Figure 3.8 shows a screenshot of the FALCON widget. The launch predictor updates the live tile to show the highest likelihood candidate apps.

3.7 Evaluation

In this section, we evaluate the performance of FALCON through extensive data analysis as well as measurements of our prototyped system implementation.

3.7.1 Dataset

The primary dataset in our analysis is the Rice LiveLab user study [97]. In this study, thirty four volunteers were given iPhone 3GSs for a period of up to fourteen months. We use two traces from the LiveLab users study: the app usage trace which records the start time and duration of every app usage, and the location trace which records the user's location with an interval of five minutes when GPS is available. By joining the app usage table and location table, we have a trace of the time and location information corresponding to each app usage. Note that since location is obtained at a relatively coarse granularity compared to app usage data, there are several instances where we do not have data at precisely the times that an app was used. To address this, we interpolate location points using a nearest neighbor method. In total, the interpolated trace of time and location information contains over 400 thousand samples across fourteen months. The majority of our evaluation is conducted based on the LiveLab dataset, except for microbenchmarks which are measured on the Windows Phone platform.



(a) Top 5 triggers across all users.

(b) Top 5 triggers for a specific user.

(c) Top 5 triggers for "Angry Birds".

Figure 3.10. Different set of session triggers for (a) aggregation across multiple users, (b) a specific user, and (c) a specific app across different users. Y-axis shows the probability that each app acts as a trigger. App name abbreviations in the graphs are: "WordsWF" = "WordsWithFriendsFree," "CyaniAH" = "CyanideAndHappiness."

3.7.2 Microbenchmarks

Before measuring the performance of FALCON, we first benchmark three critical parameters for each app: a) the time taken for launching an app, b) the amount of energy consumed during app launch, and c) the memory consumed by the app at launch time. Note that we focus on these parameters at launch time as opposed to how they vary during app usage.

Our measurements are conducted on an LG Mazza phone running Windows Phone 7.5. We measured the top twenty most popular apps in the Windows Phone Marketplace, as well as the first party apps that are preinstalled on the phone, including People, Email, and Maps.

Figure 3.9(a) shows the CDF of app launch time with and without prelaunching. The launch time includes both the OS loading time and network fetch time. The network fetch time is the average over a number of measurements under a good WiFi connection, which is an optimistic estimate since a cellular data connection is likely to be slower. Without preloading, half of the apps require more than 10 seconds to be fully launched, and around 10% of apps have a launch time of more than 20 seconds. With preloading, the app launch time can be reduced dramatically to close to zero.

Figure 3.9(b) shows the CDF of energy cost to launch an app. The energy cost as measured with a Monsoon power meter ranges from 500 to 2500 μAh, with a median energy consumption of 750 μAh. For reference, a typical LiOn battery on a mobile phone has a capacity of 1500 mAh, therefore the energy cost ranges from 0.03% to 0.17% of the battery capacity for a *single launch*. Clearly, when app usage is high, the total energy cost of launching applications can be a significant fraction of the overall budget. A benefit of FALCON is that we can cap the total amount of energy for preloading to ensure that its effect on battery life is bounded.

Figure 3.9(c) shows the CDF of memory cost for launching an app. The memory consumption varies from 10 MB to 90 MB, with a median memory consumption of 30MB. This is a significant fraction of available memory on smartphones, which typically have memory capacity between a few tens to hundreds of MB. (Our latest WP phone has about 150MB available for user apps.)

3.7.3 Benefits of Individual Features

At the core of FALCON are three features: session triggers and followers, temporal bursts, and location clusters. We now look at the benefits of using each of these features in isolation.

3.7.3.1 Session Triggers and Followers

Our goal in this section is to understand trigger and follower behavior across different users and applications. Specifically, we ask three questions: a) what applications are the best triggers across different users, b) are there differences between individuals in their trigger and follower behavior, and c) for a single user, are there differences in triggers for different follower apps?

Figure 3.10(a) shows the top 5 session triggers across all users. Not surprisingly, this list includes Email, Facebook and Safari — all popular apps. What is surprising, however, is that SMS and Phone calls are excellent triggers as well. In other words, receiving an SMS or a phone call creates an interruption into an individual's routine, and once interrupted the users tend to use other applications before they close the phone.

Figure 3.10(b) shows the top 5 triggers across all users are not necessarily the best triggers for an individual. This graph shows the best triggers for a user who is heavy user of games — in this case, the best trigger is in fact the *WordsWithFriends* game. This indicates that personalization is needed for the user, and we cannot assume that triggers are the same across everyone.

To understand if different apps have different triggers, we look at the triggers for a specific application across all users. We choose *Angry Birds* as our target app in this evaluation since it is very popular across multiple users and its loading time is representative for games. Figure 3.10(c) shows that triggers for *AngryBirds* are also different from those across all applications, suggesting that triggers should be chosen differently for each application.

To conclude, this evaluation shows that triggers need to be both user- and app-specific, and it makes the case for dynamic personalized triggers that we will evaluate in §3.7.4.2.

3.7.3.2 Temporal Bursts

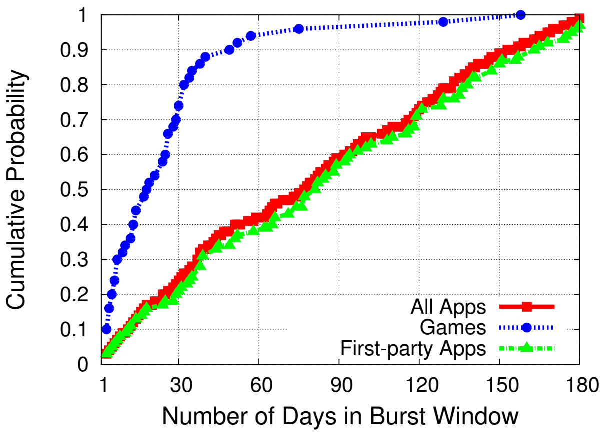


Figure 3.11. CDF of number of days that involve bursty usage for each application. 80% of games have less than 30 bursty days over 14 months usage.

We now turn to evaluate the performance of the temporal feature, namely bursty app usage. Temporal bursts can be an effective feature for improving prelaunch accuracy since it indicates when an application is heavily used during a relatively short window of time, such as a span of several days. Our evaluation looks at which applications are likely to be used in a bursty manner and which are not. Intuitively, one would expect that games are likely to be used in a bursty manner whereas First-party apps such as Email or People (WP Facebook client) should have a more steady usage pattern. Therefore, we explore the burstiness for three categories of apps — All applications, Games, and First-party apps.

Figure 3.11 shows the CDF of number of bursty days for each app category. Each point in this graph corresponds to a {user, app} pair. From this graph, we can conclude that games have stronger bursty behavior than apps such as Email and Facebook. Around 80% of games are used within 30 days out of 14 months trace length, while only 20% of the first party apps are used within the same time frame. This observation suggests that bursty behavior is a discriminating feature that we can exploit to improve the prelaunching performance, especially for games.

3.7.3.3 Location Clusters

After temporal burst feature, we now look at the performance of location clusters. Similar with temporal features, we look for applications that are heavily used within certain location clusters to improve prelaunch accuracy. We evaluate the distribution of number of location clusters per application (determined via k-means clustering). We use the same categories as the previous section — All applications, Games, and First-party apps. Our intuition is that games tend to be used in fewer locations than other apps.

Figure 3.12 shows the CDF of number of clusters where each app category is used. From this graph, we can see that games have a higher locality than other apps — 90% of game apps are used in less than 25 clusters, and more than half of games are used in less than 10 clusters. Thus, we conclude that location clustering is also a discriminating feature, particularly for games.

3.7.4 Combining Features

We now look at the benefits of combining the features of temporal burst and location clusters on prediction accuracy. Our evaluation consists of two steps. First, we look at how adding temporal features and location clusters improves accuracy

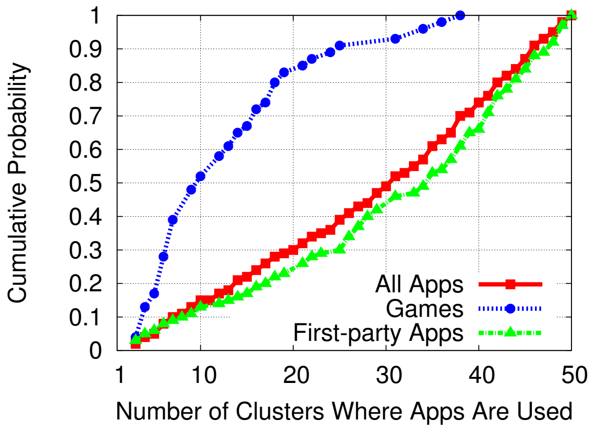


Figure 3.12. CDF of number of location clusters per app. Over 50% of games are used within less than ten clusters.

when the set of triggers are fixed across all users and apps. Second, we evaluate the benefits of making the triggers dynamic, in which case triggers can be modified within a cluster and burst window based on the individual and target application for each user.

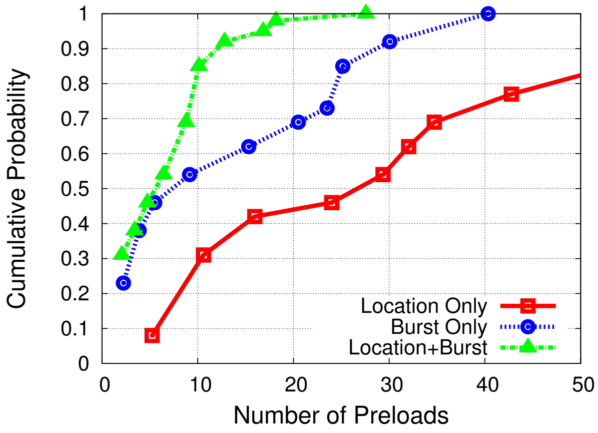


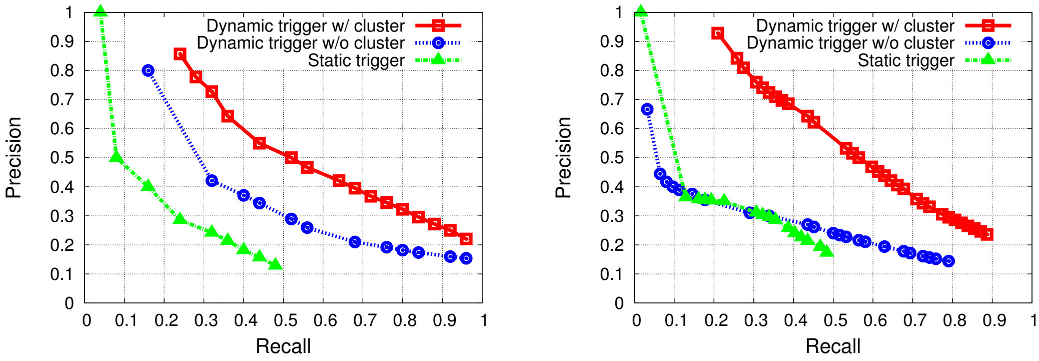
Figure 3.13. Benefit of location and temporal burst features, measured in terms of number of preloads per correct usage.

3.7.4.1 Benefits of location + temporal features

In this experiment, we look at the benefits of combining location and temporal features. We fix trigger apps to include the most common triggers across users, Phone, SMS, and Safari. We then explore the benefit of using only location clusters, only temporal bursts, and finally a combination of the two for each follower app. For each setting, we look at improvement in prelaunch accuracy, specifically, the number of preloads per app usage for each user.

Figure 3.13 shows the CDF of number of preloads across all users. Note that all curves have the same set of triggers. As seen from the figure, location clustering by itself performs poorly — the median number of preloads that lead to an actual app usage is around 27. If we use only temporal bursts, the median preload number is reduced to only six. Combining the two features gives even better performance — median number of preloads reduces to around five.

3.7.4.2 Benefits of dynamic triggers



(a) User 1

(b) User 2

Figure 3.14. Precision and recall of prelaunching a follower app for two individual users under static and dynamic trigger settings. In this graph, “Angry Birds” is used as a representative follower app that exhibits both temporal and spatial selectivity.

While combining location and temporal features improves preload accuracy, it still incurs the cost of several preloads per app usage. We now look at how dynamic triggers improve performance over static triggers. This experiment evaluates improvement in precision and recall by personalizing triggers for each user based on both location and temporal burst contexts. We focus on “Angry Birds”, a popular game that provides a representative example of an application that exhibits both location and spatial selectivity.

We compare the precision and recall of preloading under three trigger settings: static triggers for all users and location clusters, dynamic triggers for each user but the same for all location clusters, and dynamic triggers for each user at each location clusters. Figure 3.14(a) and Figure 3.14(b) shows the precision and recall for two different users when the follower app is *AngryBirds*. For both users, dynamic triggers for each location cluster achieves the best precision and recall, and clearly provides dramatic improvements over the other two cases. In addition, the performance of dynamic triggers for each location cluster is also stable for different users, while the performance of the other two schemes varies significantly for different users. This observation clearly indicates that we should use dynamic personalization of triggers at the location cluster level.

3.7.5 Evaluation of cost-benefit learner

Having comprehensively evaluated how features can enable better preloading, we turn to evaluate the performance of Cost-Benefit Learner (CBL). Our cost-benefit analysis includes information about the cost of a preload in terms of energy, and the benefits of the preload in terms of app loading latency, or in the case of network prefetching, in terms of the freshness of fetched data. We use the energy/latency benchmarks from §3.7.2 in this evaluation. We assume for simplicity that each app has equal memory consumption.

3.7.5.1 Performance of Prefetching

In this experiment, we look at the benefit for apps that require fetching new content during the launch process. We take the Email application as a representative app, which is both heavily used and incurs significant delay for fetching emails from servers.

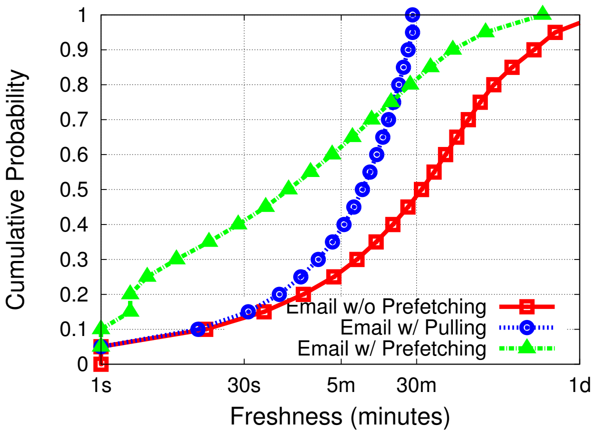


Figure 3.15. Benefit of prelaunching for app freshness. Email is used as a representative application that is both heavily used and has high network content fetch time.

First, we compare CBL (noted as Email w/ Prefetching) against two baseline schemes: a) no prefetching (noted as Email w/o Prefetching), and b) active pull with an interval of 30 minutes(noted as Email Pulling). Our evaluation metric is freshness of the app, which is defined as the time that elapses between retrieval of content from the server and when the app is opened by the user.

Figure 3.15 shows the freshness of rhe Email app using our scheme (note that the x-axis is in log scale). Compared with no prefetching, we improve freshness significantly by reducing the median freshness from 30 minutes to around 3 minutes. Our scheme also outperforms the pulling-based scheme, whose median freshness is about 10 minutes.

3.7.5.2 Performance of Preloading

In this experiment, we look at how app loading time can be improved by our CBL learner. We compare our scheme against LRU caching, which is the standard policy on most smartphones including Windows Phone. In this experiment, we select 24 popular apps (including games, social apps, and First-party apps) as follower to cover the most common app usage cases on WP platform. We assume that the memory manager allows up to five apps, as is the case on the current Windows Phone platform. We also assume that every app has equal memory consumption.

While both CBL and LRU take advantages of bursty behavior of apps, there is a subtle but important difference in the way FALCON vs. LRU work for bursty apps. It might seem like LRU would work well when apps are bursty, but the burstiness of apps (e.g. games) is at the timescale of days, not minutes. In other words, a "bursty game" might be used once or twice a day. LRU is designed to work well when burstiness is at short timescales, where apps are so frequently used that they are not evicted. CBL is able to take advantage of burstiness at the timescale of days because we are looking at longer term contextual and session behavior.

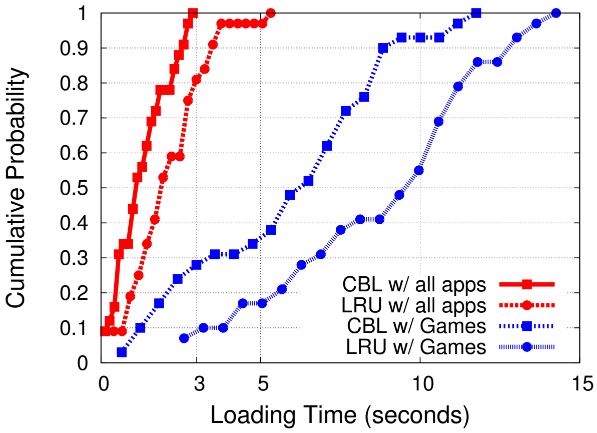


Figure 3.16. Benefit of loading time with prefetching. CBL is compared against LRU, both when all applications are used, and when only games are used. We look at games separately since these apps exhibit both temporal burstiness and spatially correlated usage patterns.

Our CBL outperforms LRU by up to 100% in terms of loading time. Figure 3.16 shows the median loading time is around 1 second and 2 seconds for CBL and LRU, respectively. Even when LRU already performs well in terms of loading time, our scheme still improves the loading time by half. For games, the median loading time is also reduced from 9 seconds with LRU, to 6 second with CBL.

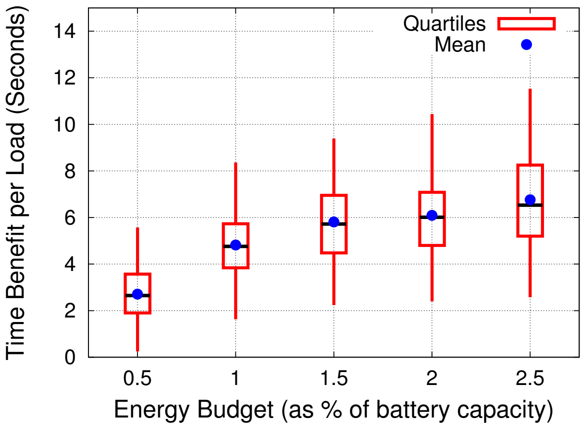


Figure 3.17. Energy cost and launching time benefit of using the combination of dynamic triggers, temporal bursts, and location clustering for prelaunch.

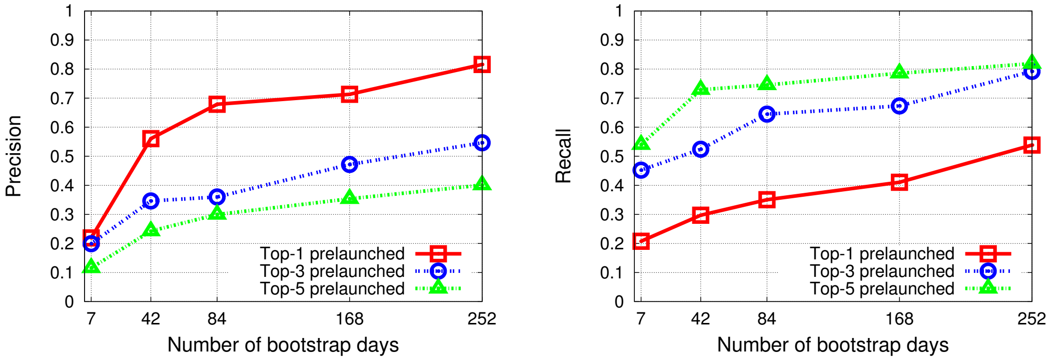
3.7.5.3 Overall benefits

Figure 3.17 shows the aggregated cost and benefit of using CBL. The x-axis of this graph represents the energy budget provided to CBL, and is normalized to the percentage of a fully charged phone battery(1500 mAh) as the daily budget. The y-axis represents the benefit of loading time which includes both app loading time and content fetching time. Our estimate for content fetching time is conservative since we used our micro-benchmark results under good WiFi connections. Our target metric is the reduced loading time per app usage. Figure 3.17 shows that with CBL, the average loading time for an app is reduced to 5.5 seconds with the energy cost of less than 1% battery power, and less than 6 seconds with 2% battery power.

3.7.6 Bootstrapping FALCON

Having evaluated the overall performance of cost-benefit learner of FALCON, we now ask how fast the cost-benefit learner can learn from history of app usage to make accurate prelaunch decisions. In this experiment, we tune the training dataset size by progressively adding new app usage data to the set, and study the performance of CBL in terms of precision and recall aggregated across all follower candidates.

We split the 14-month long app usage trace as follows in this experiment: we use the data from day one to day N as training data, and we tune the number N such that the training dataset size is set from 1% of total dataset size to 60%. We then use the 20% of data immediately following the training dataset as the testing dataset. For each setting, both location clustering and temporal burst windows are re-calculated to reflect temporal dynamics in features. To simplify the analysis, we do not consider energy constraint in this experiment but only study precision and recall of up to the top five follower candidates.



(a) Precision

(b) Recall

Figure 3.18. Precision and recall of bootstrapping.

From Figure 3.18(a), we conclude that the topmost candidate from FALCON begins to reach over 50% prediction precision with around 35 days of training data. When the training dataset size grows to 84 days, or 2.8 months, prediction precision is close to 70%. The precision climbs to over 80% when the training dataset size is over 252 days, or 8 months. Figure3.18(b) shows that the recall of FALCON ins improved as the fraction of training set increases. If we prelaunch the top 5 follower candidates, over 70% of the targeted apps can be successfully preloaded by FALCON when the training dataset is 42 days long. The recall increases to over 80% when the training dataset size is more than 252 days, or 8 months long.

Binary Size	129 KB
Memory (stable state)	1840 KB
Processor utilization (stable state)	<1%
Processor utilization per prediction	<3%
Energy cost per prediction	< 3 μAh

Table 3.2. FALCON Overhead Profile

In conclusion, the performance of FALCON grows as the training data size increases. Even with a relatively small training dataset of two months, FALCON can make prelaunch decisions with good accuracy, demonstrating that good performance can be achieved with a few months of training.

3.7.7 System Overhead

The system overhead of our FALCON implementation is very modest. Table 3.2 shows its resource consumption profile. Memory usage during stable state is less than 2MB, and does not increase measurably during prediction. Processor utilization is low during stable state, and reaches up to 3% during prediction. We also measured the energy overhead of FALCON predictions with a Monsoon power meter, and we observed that the extra cost FALCON incurs for each launch prediction is only a few μAh.

Note that we have not accounted for the periodic geolocation sampling costs as a component of the overhead. This is for two reasons: a) we don't need GPS geolocation since we only need cluster-level information, hence cell tower triangulation will suffice, and b) we can be opportunistic about obtaining GPS readings by leveraging cached geo-location results generated from other app's requests. In this manner, geo-location information can be obtained with low cost.

Another component of our system is online feature extraction. Our current implementation of location clustering is relatively expensive to perform on a mobile phone, and therefore needs to be done in the cloud. In contrast, the burst detection can be done online with a slightly modified version of the wavelet-based burst detection algorithm described in §3.5. We implemented a light-weight online burst window detection algorithm that adapts the sliding window size based on historical data, and uses the window to detect bursts. Our measurements indicate that our online burst detection has a small performance loss compared to the wavelet based scheme as presented in [134], with 4% less precision, and 10% less recall.

3.8 Discussion and Conclusion

Mobile device usage is inherently brief, competing with myriad other real demands on human attention. As such, it is important that app interactions are rapid and responsive. Despite the fact that mobile devices are becoming increasingly resource rich, launch times are and will remain a significant challenge. This is due to two reasons. First, network-bound apps will continue to be limited by network fetch latency, which is constrained by cellular technology. Second, increasingly graphics- and media-rich apps targeting up-market phones will perpetually leave previous-generation and budget smartphones feeling sluggish to launch apps.

Our work addresses this important problem through the design of FALCON, a system that uses location and temporal contexts to predictively launch applications, thereby reducing perceived delay for users. We designed a set of features to capture the essential characteristics of app access patterns, and a launch predictor to adaptively balances latency reduction benefit with energy launch costs. We have shown that launch prediction using context can yield latency reductions of around 50% vs. LRU at a manageable 2% energy cost, and present fresh content at launch within the last 3 minutes. The FALCON modification to the Windows Phone OS consists of a context source manager, embedded launch predictor and prelaunch event upcall dispatcher. These elements are sufficiently low overhead to require only 2MB memory and negligible energy and processing.

Although our results show dramatic benefits of FALCON, there are several possible improvements and opportunities that we have not explored. One area is eliminating reliance on external servers or cloud services for model training in FALCON, thereby mitigating the need for sharing personal data with external service providers. An additional benefit of reducing reliance on the cloud is faster adaptation to underlying changes in app usage habits.

Another open question is how users' expectations will change as the OS predictively prelaunches apps on their behalf. Will the fact that prelaunching can result in variable launch time hinder usability of mobile applications or will users appreciate the quicker interactions that are made possible by such techniques? We will address these questions in future studies.

CHAPTER 4

CROWDSEARCH: ACCURATE AND REAL-TIME IMAGE SEARCH FOR MOBILE PHONES BY EXPLOITING CROWDSOURCING

4.1 Introduction

In this chapter, we study the problem of how to exploit crowdsourcing-based human validation to improve the accuracy of cloud-based image search for mobile phones, and how to providing realtime search performance despite having human involved in the process.

Mobile phones are becoming increasingly sophisticated with a rich set of on-board sensors and ubiquitous wireless connectivity. One of the primary advantages of ubiquitous internet access is the ability to search anytime and anywhere. It is estimated that more than 70% of smart phone users perform internet search [112]. The growth rate in mobile internet search suggests that the number of search queries from phones will soon outstrip all other computing devices.

Search from mobile phones presents several challenges due to their small form-factor and resource limitations. First, typing on a phone is cumbersome, and has led to efforts to incorporate on-board sensors to enable multimedia search. While the use of GPS and voice for search is becoming more commonplace, image search has lagged behind. Image search presents significant challenges due to variations in lighting, texture, type of features, image quality, and other factors. As a result, even state-of-art image search systems, such as Google Goggle [113], acknowledge that they only work with certain categories of images, such as buildings. Second, scrolling through multiple pages of search results is inconvenient on a small screen. This makes it important for search to be *precise* and generate few erroneous results. Third, multimedia searches, particularly those using images and video clips, require significant memory, storage, and computing resources. While remote servers, for example cloud computing infrastructures [26], can be used to perform search, transferring large images through wireless networks incurs significant energy cost.

While automated image search has limitations in terms of accuracy, humans are naturally good at distinguishing images. As a consequence, many systems routinely use humans to tag or annotate images (e.g. Google Image Labeler [111]). However, these systems use humans for tagging a large corpus of image data over many months, whereas search from phones needs fast responses within minutes, if not seconds.

In this chapter we present CrowdSearch, an accurate image search system for mobile phones. CrowdSearch combines automated image search with *real-time* human validation of search results. Automated image search uses a combination of local processing on mobile phones and remote processing on powerful servers. For a query image, this process generates a set of candidate search results that are packaged into tasks for validation by humans. Real-time validation uses the Amazon Mechanical Turk (AMT), where tens of thousands of people are available to work on simple tasks for monetary rewards. Search results that have been validated are returned to the user.

4.1.1 Challenges in Designing Image Search Engines with Humans-in-the-loop

The combination of automated search and human validation presents a complex set of tradeoffs involving delay, accuracy, monetary cost, and energy. From a crowdsourcing perspective, the key tradeoffs involve delay, accuracy and cost. A single validation response is often insufficient since 1) humans have error and bias, and 2) delay may be high if the individual person who has selected the task happens to be slow. While using multiple people for each validation task and aggregating the results can reduce error and delay, it incurs more monetary cost. From an automated image search perspective, the tradeoffs involve energy, delay, and accuracy. Exploiting local image search on mobile phones is efficient energy-wise but the resource constraints on the device limits accuracy and increases delay. In contrast, remote processing on powerful servers is fast and more accurate, but transmitting large raw images from a mobile phone consumes time and energy.

CrowdSearch addresses these challenges using three novel ideas. First, we develop accurate models of the delay-accuracy-cost behavior of crowdsourcing users. This study is a first-of-its-kind, and we show that simple and elegant models can accurately capture the behavior of these complex systems. Second, we use the model to develop a predictive algorithm that determines which image search results need to be validated, when to validate the tasks, and how to price them. The algorithm dynamically makes these decisions based on the deadline requirements of image search queries as well as the behavior observed from recent validation results. Third, we describe methods for partitioning of automated image search between mobile phones and remote servers. This technique takes into account connectivity states of mobile phones (3G vs WiFi) as well as search accuracy requirements.

4.2 Related Work

In this section, we provide an overview of closely related work on crowdsourcing and image search for mobile phones.

4.2.1 Participatory Sensing using Images

Sensing using mobile phones has become popular in recent years (e.g. Urban sensing [17], Nokia's Sensor Planet [114], MetroSense [31]). The emphasis of such efforts is to utilize humans with mobile phones for providing sensor data that can be used for applications ranging from traffic monitoring to community cleaning. Our work is distinct from these approaches in that we focus on designing human-in-the-loop computation systems rather than just using phones for data collection.

Research in image search straddles advances in image processing and information retrieval. Techniques that we use in our system, such as SIFT, visterm-extraction using vocabulary trees, and inverted index lookup are based on state-of-art approaches in this area [61, 87, 23, 85]. In our prior work, we have also applied such techniques in the context of sensor networks [130]. However, a well-known limitation is that these techniques work best for mostly planar images such as buildings, and are poor for non-planar images such as faces. This limitation is a reason why the recently released Google Goggle [113] system is primarily advertised for building landmarks. While our use of real-time human validation doesn't improve the performance of an image search engine, it can help filter incorrect responses to return only the good ones.

Much recent work such as [64, 11] has looked at mobile sensing and people centric sensing. Of relevance to our work is the iScope system [133], which is a multi-modal image search system for mobile devices. iScope performs image search using a mixture of features as well as temporal or spatial information where available. While using multiple features can help image search engine performance, it does not solve the problem of low accuracy for certain categories. The crowdsourcing part of CrowdSearch is complementary to iScope, Google Goggles, or any other image search engine whose focus is on improving the performance of automated search. Any of these techniques can be augmented with CrowdSearch to achieve close to 100% precision at low cost.

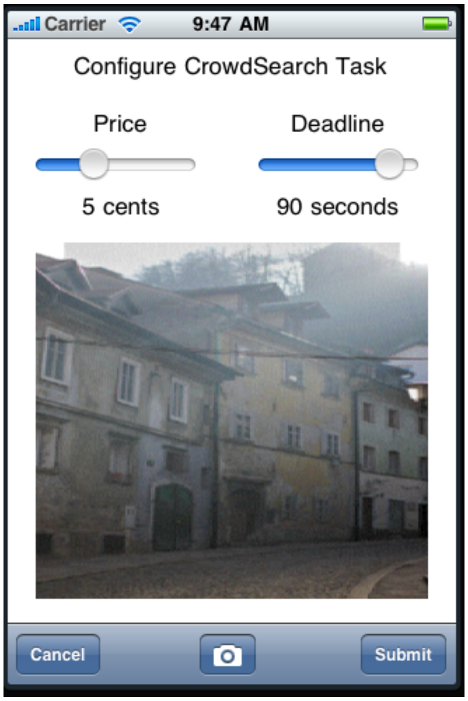
4.2.2 Online Crowdsourcing Services

There have been numerous crowdsourcing services that exploits a large number of online people to provide services that requires human intelligence. Luis Von Ahn's pioneering work on reCaptcha [121] uses humans to solve difficult OCR tasks, thereby enabling digitization of old books and newspapers. His another work on the ESP game [120] uses humans for finding good labels for images, thereby facilitating image search. The two systems use different incentive models — reCaptcha protects websites against robots, whereas ESP rewards participants with points if the players provide matching labels. Other popular crowdsourcing models include the auction-based crowdsourcing (e.g. Taskcn [115]), and simultaneous crowdsourcing contests (e.g. TopCoder [116]). Our work is inspired by these approaches, but differs in that we focus on using crowdsourcing to provide real-time search services for mobile users. To our knowledge, this model has not been explored prior to our work.

Many applications have begun to utilize micro-payment crowdsourcing systems such as AMT. This includes the use of crowdsourcing for labelling images and other complex data items [53, 101, 96]. For example, Sorokin et al. [101] show that using AMT for image annotation is a quick way to annotate large image databases. However, this work is done in an offline manner and not in the context of a real-time search system. We also find that image annotation is quite noisy in comparison to validation of candidates from a search engine.

A central contribution of our work is modeling delay from crowdsourcing of validation tasks. While there have been several studies to understand quality of results from AMT solvers [53] [96], we are not aware of any other work that attempts to model delay from the system. We believe that our models can have broader applicability for other applications that use crowdsourcing systems.

Amazon Remembers [118] is an application that takes phone-based queries and uses crowdsourcing for retrieving product information from the queries. While Ama-


Configure CrowdSearch Task

Price: 5 cents

Deadline: 90 seconds

Figure 4.1. An iPhone interface for CrowdSearch system. A building image is captured by user and sent to CrowdSearch as a search query. Users can specify the price and deadline for this query.

zon Remembers combines mobile phones with crowdsourcing, CrowdSearch is considerably more sophisticated in its use of crowdsourcing since it enables *real-time* responses by specifying deadlines and combines automated and human processing.

4.3 CrowdSearch System Overview

In this section, we provide an overview of the major workflow of our CrowdSearch system.

CrowdSearch comprises three components: (i) the mobile phone, which initiates queries, displays responses, and performs local image processing, (ii) a powerful remote server, or cloud computing backend, which performs automated image search, and triggers image validation tasks, and (iii) a crowdsourcing system that validates image search results.

System Operation: CrowdSearch requires three pieces of information prior to initiating search: (a) an image query, (b) a query deadline, and (c) a payment mechanism for human validators. There are several options for each of these components. An image query may be either a single image or accompanied by other modalities, such as GPS location or text tags. The deadline can be either provided explicitly by the user using a micropayment account such as Paypal, or a search provider, such as Google Goggle, who pays for human validation to improve performance and attract more customers.

In the rest of this chapter, we assume that a search query comprises solely of an image, and is associated with a deadline as well as payment mechanism. Given such a query, CrowdSearch attempts to return at least one correct response prior to the deadline while minimizing monetary cost.

The major operations of CrowdSearch comprises of two steps. The first step is processing the image query using an automatic image search engine. The search engine searches through a database of labeled images and returns a ranked list of candidate results. Despite the fact that search is initiated by images from mobile phones, the database of labeled images can include images obtained from diverse sources including FlickR or Google. The downside of automated image search is that many results may be incorrect, particularly for non-planar objects such as faces.

The second step in our system is the CrowdSearch algorithm that uses a crowdsourcing system to validate the candidate results. In particular, we use the Amazon Mechanical Turk (AMT), since it has APIs for automatic task posting, and has a large user base of tens of thousands of people. Human validation is simple — for each <query image, candidate image> pair, a human validator just clicks on YES if they match and NO if they don't match. Only thumbnails of the images are sent for validation to minimize communication required from the phone. In return for the answer, the validator is paid a small sum of money as reward (one or few cents). The

Crowdsearch algorithm considers several tradeoffs while determining how candidate images are validated. Since human validation consumes monetary cost, it attempts to minimize the number of human validators that need to be paid to get a good response for a user query. However, CrowdSearch also needs to take into account the deadline requirements as well as the need to remove human error and bias to maximize accuracy. To balance these tradeoffs, CrowdSearch uses an adaptive algorithm that uses delay and result prediction models of human responses to judiciously use human validation. Once a candidate image is validated, it is returned to the user as a valid search result.

4.4 Exploiting Crowdsourcing for Search Results Validation

In this section, we first provide a background of the Amazon Mechanical Turk (AMT). We then discuss several design choices that we make while using crowdsourcing for image validation including: 1) how to construct tasks such that they are likely to be answered quickly, 2) how to minimize human error and bias, and 3) how to price a validation task to minimize delay.

Background: We now provide a short primer on the AMT, the crowdsourcing system that we use in this work. AMT is a large-scale crowdsourcing system that has tens of thousands of validators at any time. The key benefit of AMT is that it provides public APIs for automatic posting of tasks and retrieval of results. The AMT APIs enable us to post tasks and specify two parameters: (a) the number of duplicates, *i.e.* the number of independent validators who we want to work on the particular task, and (b) the reward that a validator obtains for providing responses. A validator works in two phases: (a) they first accept a task once they identify that they would like to work on it, which in turn decrements the number of available duplicates, and (b) once accepted, they need to provide a response within a period specified by the task.

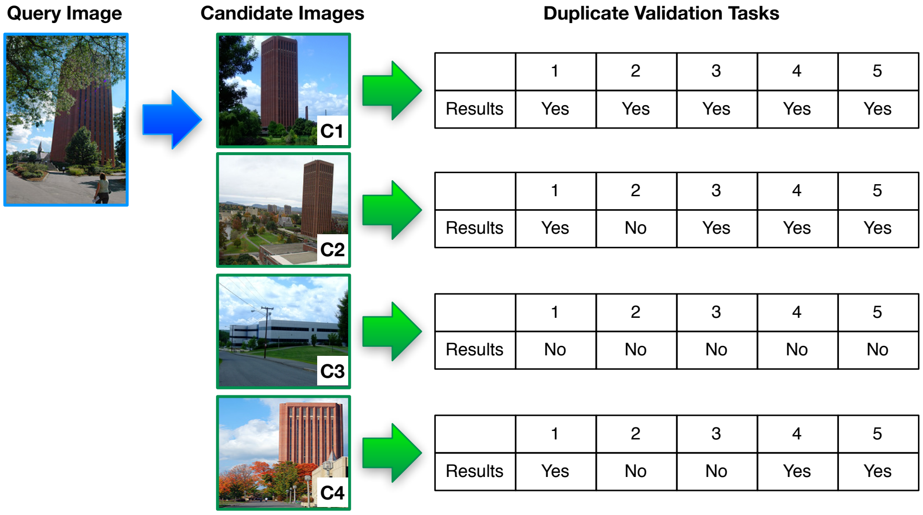
One constraint of the AMT that pertains to CrowdSearch is that the number of duplicates and reward for a task that has been posted cannot be changed at a later point. We take this practical limitation in mind in designing our system.

Constructing Validation Tasks: How can we construct validation tasks such that they are answered quickly? Our experience with AMT revealed several insights. First, we observed that asking people to tag query images and candidate images directly is not useful since: 1) text tags from crowdsourcing systems are often ambiguous and meaningless (similar conclusions have been reported by other crowdsourcing studies [53]), and 2) tasks involving tagging are unpopular, hence they incur large delay. Second, we found that having a large validation task that presents a number of <query image, candidate image> pairs enlarges human error and bias since a single individual can bias a large fraction of the validation results.

We settled on an a simple format for validation tasks. Each <query image, candidate image> pair is packaged into a task, and a validator is required to provide a simple YES or NO answer: YES if the two images are correctly matched, and NO otherwise. We find that these tasks are often the most popular among validators on AMT.

Minimizing Human Bias and Error: Human error and bias is inevitable in validation results, therefore a central challenge is eliminating human error to achieve high accuracy. We use a simple strategy to deal with this problem: we request several duplicate responses for a validation task from multiple validators, and aggregate the responses using a majority rule. Since AMT does not allow us to dynamically change the number of duplicates for a task, we fix this number for all tasks. In §4.7.2, we evaluate several aggregation approaches, and show that a majority of five duplicates is the best strategy and consistently achieves us more than 95% search accuracy.

Pricing Validation Tasks: Crowdsourcing systems allow us to set a monetary reward for each task. Intuitively, a higher price provides more incentive for human



	1	2	3	4	5
Results	Yes	Yes	Yes	Yes	Yes

	1	2	3	4	5
Results	Yes	No	Yes	Yes	Yes

	1	2	3	4	5
Results	No	No	No	No	No

	1	2	3	4	5
Results	Yes	No	No	Yes	Yes

Figure 4.2. Shown are an image search query, candidate images, and duplicate validation results. Each validation task is a Yes/No question about whether the query image and candidate image contains the same object.

validators, and therefore can lead to lower delay. This raises the following question: *is it better to spend X cents on a single validation task or to spread it across X validation tasks of price one cent each?* We find that it is typically better to have more tasks at a low price than fewer tasks at a high price. There are three reasons for this behavior: 1) since a large fraction of tasks on the AMT offer a reward of only one cent, the expectation of users is that most tasks are quick and low-cost, 2) crowdsourcing systems like the AMT have tens of thousands of human validators, hence posting more tasks reduces the impact of a slow human validator on overall delay, and 3) more responses allows better aggregation to avoid human error and bias. Our experiments with AMT show that the first response in five one cent tasks is 50 - 60% faster than a single five cent task, confirming the intuition that delay is lower when more low-priced tasks are posted.

4.5 CrowdSearch Algorithm

Given a query image and a ranked list of candidate images, the goal of human validation is to identify the correct candidate images from the ranked list. Human validation improves search accuracy, but incurs monetary cost and human processing delay. We first discuss these tradeoffs and then describe how CrowdSearch optimizes overall cost while returning at least one valid candidate image within a user-specified deadline.

4.5.1 Delay-Cost Tradeoffs

Before presenting the CrowdSearch algorithm, we illustrate the tradeoff between delay and cost by discussing posting schemes that optimize one or the other but not both.

Parallel posting to optimize delay: A scheme that optimizes delay would post all candidate images to the crowdsourcing system at the same time. (We refer to this as *parallel posting.*) While parallel posting reduces delay, it is expensive in terms of monetary cost. Figure 4.2 shows an instance where the image search engine returns four candidate images, and each candidate image is validated by five independent validators. If each of these validators is paid a penny, the overall cost for validating all responses would be 20 cents, a steep price for a single search. Parallel posting is also wasteful since it ignores the fact that images with higher rank are more likely to be better matches than those lower-ranked ones. As a result, if the first candidate image is accurate, the rest of the candidates need not to be posted.

Serial posting to optimize cost: In contrast to parallel posting, a scheme that optimizes solely the monetary cost would post tasks *serially.* A serial posting scheme first posts the top-ranked candidate for validation, and waits to see if the majority of validators agree that it is a positive match. If so, the process ends and returns a positive match, otherwise the next candidate is posted, and so on. This scheme uses the least number of tasks to find the first correct match, and thus costs considerably less than the parallel posting scheme. Taking Figure 4.2 as an example, the serial posting scheme would just cost five cents, since the first candidate is a valid match. However, in cases where top-ranked image is incorrect, serial posting incurs much higher delay than parallel posting.

Clearly, parallel and serial posting are two extreme schemes that sacrifice either cost or delay. Instead, we propose an algorithm that achieves a tradeoff between these two metrics.

4.5.2 Optimizing Delay and Cost

Key Insight: CrowdSearch tries to provide a balance between serial and parallel schemes. More precisely, the goal is to minimize monetary cost while ensuring that *at least one valid candidate*, if present in the ranked list returned from the search engine, is provided to the user within a user specified deadline. If all candidates images are incorrect, no response is returned to the user. For example, in Figure 4.2, candidates C_1, C_2, and C_4 are all valid responses, hence it is sufficient to validate any one of them and return it as a response to the mobile user. Intuitively, such an objective also fits well with image search on mobile phones, where few good results are more appropriate for display.

To illustrate the main idea in CrowdSearch, consider the case where we have one query image and five candidate images returned by the image search engine. Assume that we posted the top-ranked candidate, C_1, as a validation task at time $t = 0$ and the user-specified deadline is time $t = 2$ *mins*. At time $t = 1$ *min*, say that two responses have been received for this task, a Yes and a No (i.e. sequence 'YN'). Within the next 1 *min*, there are multiple possible incoming responses, such as 'YY', 'YNY', and others, that can lead to a positive validation result under majority(5) rule. The CrowdSearch algorithm estimates the probability that any one of these valid sequences occurs during the next minute. This estimation is done by using: 1) models of inter-arrival times of responses from human validators, and 2) probability estimates for each sequence that can lead to a positive validation result. If the probability of a valid result within the deadline is less than a pre-defined threshold P_{TH}, for instance 0.6, CrowdSearch posts a validation task for the next candidate image to improve the chance of getting at least one correct match. The above procedure is performed at frequent time-steps until either the deadline is reached or all candidates have been posted for validation.

The Algorithm Having explained the key insight, we now present the complete CrowdSearch prediction algorithm as shown in Algorithm 1. This algorithm handles all the ongoing tasks, each task has received partial results from validators. For each task T_x, let S_i denote the partial sequence received. For instance, in the example above, two results of 'YN' have been received. The algorithm computes the probability that at least one of the ongoing tasks is going to have a positive answer under the majority(N) rule, where N is the number of duplicates for each validation task. For each task T_x, the CrowdSearch algorithm traverses all possible sequence of results S_j that begin with the received sequence S_i. If the sequence S_j leads to a positive answer, CrowdSearch predicts the probability that the remaining results can be received before the deadline, as shown in line 4-6 in algorithm 1. Here CrowdSearch calls two functions, *DelayPredict* and *ResultPredict*. The former function estimates the probability that sequence S_j will be received before the deadline, and the latter estimates the probability that the sequence S_j will occur given that sequence S_i has been received so far. We assume that these two probabilities are independent to each other since the delay and content of a result sequence have no clear causal relation. Thus, the product of the two, P_j, is the probability that the sequence S_j is received prior to the deadline. We compute P_+, which is the accumulation of P_j for all cases where the remaining sequence leads to a positive answer. This gives us the predicted

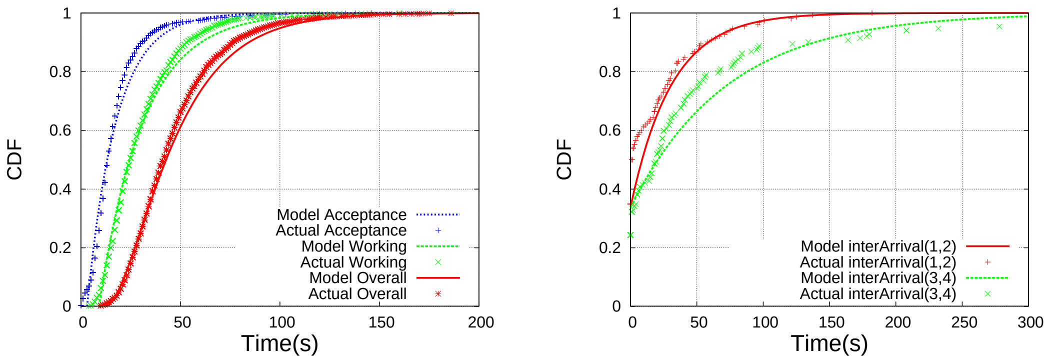
Algorithm 1 CrowdSearch()

```
 1: $P_{fail} \leftarrow 1$
 2: for each ongoing task $T_x$ do
 3:    $S_i \leftarrow$ sequence of results received
 4:    $P_+ \leftarrow 0$
 5:    for each $S_j$ that starts with $S_i$ do
 6:       $P_{delay} \leftarrow$ DelayPredict($|S_i|$, $|S_j|$)
 7:       $P_{results} \leftarrow$ ResultPredict($S_i$, $S_j$)
 8:       $P_j \leftarrow P_{results} \times P_{delay}$
 9:       if $Majority(S_j) = Yes$ then
10:          $P_+ \leftarrow P_+ + P_j$
11:       end if
12:    end for
13:    $P_{fail} \leftarrow P_{fail} \times (1 - P_+)$
14: end for
15: $P_{suc} \leftarrow 1 - P_{fail}$
16: if $P_{suc} \geq P_{TH}$ then
17:    Return True
18: else
19:    Return False
20: end if
```

probability that current task can be validated as positive given S_i results are received. Having predicted the probability for a single validation task, now we consider the case where multiple validation tasks are concurrently ongoing for a search query. Since our goal is to return at least one correct candidate image, we first compute P_{fail}, which is the probability that none of the ongoing tasks are correct. Hence $P_{suc} = 1 - P_{fail}$ is the probability we want to compute. P_{fail} is the product of the probability that each of the ongoing task fails, since all tasks are independent to each other. We compute P_{fail} as shown in line 13 in the algorithm 1. If P_{suc} is less than a pre-defined threshold P_{TH}, we post the next task, else we wait for current tasks. In the rest of this section, we discuss *DelayPredict* and *ResultPredict* that are central to our algorithm.

4.5.3 Delay Prediction Model

We now consider the problem of predicting the time that sequence S_j is received given a partial sequence S_i has been received. This problem can be split into the sum



(a) Overall delay model

(b) Inter-arrival delay model

Figure 4.3. Delay models for overall delay and inter-arrival delay. The overall delay is decoupled with acceptance and submission delay.

of inter-arrivals from S_i to S_{i+1}, S_{i+1} to S_{i+2}, and so on till S_{j-1} to S_j. We start with a model of the inter-arrival delay between any two responses obtained from validators, and then use the inter-arrival models to determine the overall delay.

4.5.3.1 Delay Modeling

We consider two cases, the first is when no responses have been received so far *i.e.* when $i = 0$, and the second when one or more responses have been received *i.e.* when $i > 0$. The two cases need to be handled differently since in the former case we model the delay for the arrival of the first response, whereas in the latter case we model the inter-arrival times between responses.

Case 1 - Delay for the first response: The delay of an AMT validator can be separated into two parts: *acceptance delay* and *submission delay*. Acceptance delay means the time from the validation task being posted to being accepted by a validators. Submission delay means the time from the task being accepted to being submitted by a validator. The total delay is the summation of acceptance delay and submission delay.

Figure 4.3(a) shows the complementary cumulative distribution function(CCDF) of our delay measurements over more than 1000 tasks. Note that the y-axis is in log scale. It is clear that both acceptance delay and submission delay follow exponential distributions with a small delay offset. The exponential behavior follows intuition due to the law of independent arrivals, and the fact that there are tens of thousands of individuals on AMT. The offset for acceptance delay results from delay in refreshing the task on the web browser, internet connection delays, and time taken by the validator to search through several thousands of tasks to locate ours. The offset for working time results from the time to read the task, input the result, and submit the answer.

Let λ_a and λ_s to represent the rate of acceptance and submissions, and c_a and c_s represent the offsets of acceptances and submissions. Their probability density function, denoted as $f_a(t)$ and $f_s(t)$, are

$$f_a(t) = \lambda_a e^{-\lambda_a(t-c_a)}$$

and

$$f_s(t) = \lambda_s e^{-\lambda_s(t-c_s)}$$

How can we obtain the overall delay distribution, which is sum of acceptance and submission delay? We first determine whether the two delays are independent of each other. Intuitively, the two should be independent since the acceptance delay depends on when a validator chanced upon our validation tasks, whereas submission delay depends on how fast the validators works on tasks. To confirm independence, we compute the Pearson's product-moment coefficient between the two delays. Our results indicate that the coefficient score is less than 0.05, which means there is no linear relation between acceptance and submission delay.

Given independence, the PDF of overall delay can be computed as the convolution of the PDFs of acceptance delay and submission delay. Let $f_o(t)$ denote the pdf of overall delay we have:

$$\begin{aligned} f_o(t) &= f_a(t) * f_s(t) \\ &= \int_{c_s}^{t-c_a} [\lambda_a e^{-\lambda_a((t-u)-c_a)}][\lambda_s e^{-\lambda_s(u-c_s)}]du \\ &= \frac{\lambda_a \lambda_s}{\lambda_a - \lambda_s}(e^{-\lambda_s(t-(c_a+c_s))} - e^{-\lambda_a(t-(c_a+c_s))}) \end{aligned}$$

Let $c = c_a + c_s$. The above expression can be written as:

$$f_o(t) = \frac{\lambda_a \lambda_s}{\lambda_a - \lambda_s}(e^{-\lambda_s(t-c)} - e^{-\lambda_a(t-c)}) \tag{4.1}$$

Case 2 - Inter-arrival delay between responses: Figure 4.3(b) shows the CCDF of inter-arrival delay between validation tasks. These delays are clearly a set of exponential distributions with different rates. The constant offsets are counteracted since we are looking at the difference between arrivals. Thus, the PDF of inter-arrival time between response i and $i+1$, denoted as $f_i(t)$, can be modeled as an exponential function:

$$f_i(t) = \lambda_i e^{-\lambda_i t} \tag{4.2}$$

4.5.3.2 Delay Prediction

Given the arrival time for the first response and the inter-arrival times between responses, we can predict delay as follows. Consider the example shown in Figure 4.4. Suppose we have received two response at 60 seconds, and the current time to the origin of the query is 75 seconds. At this time, we want to predict the probability that five responses are received within the remaining 45 seconds to the deadline. Let X_j

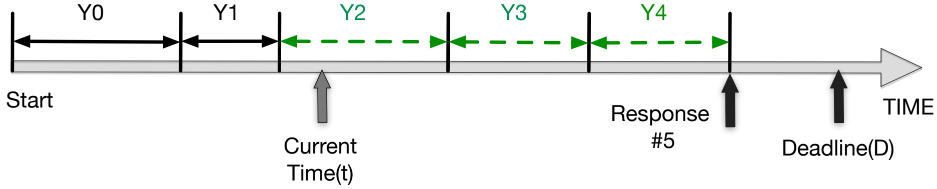


Figure 4.4. An example of predicting the delay of 5^{th} response given the time of 2^{nd} response. The delay is the summation of the inter-submission delay of Y_2, Y_3 and Y_4.

denote the time of receiving the j_{th} response, and $\hat{X}_i$ denote the tuple $X_1, X_2, \cdots, X_i$, and $\hat{t}_i$ denote the tuple $t_1, t_2, \cdots, t_i$, where t_i is the time of receiving the i^{th} response. Let D denote the deadline, t denote the current time, All times are relative to the beginning of the query. The probability we want to estimate is:

$$P_{ij} = P(X_j \leq D \mid X_{i+1} \geq t, \hat{X}_i = \hat{t}_i) \tag{4.3}$$

We consider this problem from the perspective of inter-arrival times between responses. Let $Y_{i,j}$ denote the inter-arrival time between the response i and j. This inter-arrival time is the sum of a set of inter-arrival times between adjacent responses:

$$Y_{i,j} = \sum_{k=i}^{j-1} Y_{k,k+1}$$

Therefore, P_{ij} can be presented as:

$$P_{ij} = P(Y_{i,j} \leq D - t_i \mid Y_{i,i+1} \geq t - t_i, \hat{X}_i = \hat{t}_i)$$

From our inter-arrival delay model, we know that all inter-arrival times are independent. Thus, we can present the probability density function of $Y_{i,j}$ as the convolution of the inter-arrival times of response pairs from i to j.

Before applying convolution, we first need to consider the condition $Y_{i,i+1} \geq t - t_i$. This condition can be removed by applying the law of total probability. We sum up all the possible values for $Y_{i,i+1}$, and note that the lower bound is $t - t_i$. For each $Y_{i,i+1} = t_x$, the rest part of $Y_{i,j}$, or $Y_{i+1,j}$, should be in the range of $D - t_i - t_x$. Thus the condition of $Y_{i,i+1}$ can be removed and we have:

$$P_{ij} = \sum_{t_x = t - t_i}^{D - t_i} P(Y_{i,i+1} = t_x) P(Y_{i+1,j} \leq D - t_i - t_x) \tag{4.4}$$

Now we can apply the convolution directly to $Y_{i+1,j}$. Let $f_{i,j}(t)$ denote the pdf of inter-arrival between response i and j. The pdf of $Y_{i+1,j}$ can be expressed as:

$$f_{i+1,j}(t) = (f_{i+1,i+2} * \cdots * f_{j-1,j})(t)$$

Combining this with Equation 4.4, we have:

$$P_{ij} = \sum_{t_x = t - t_i}^{D - t_i} (f_{i,i+1}(t_x) \sum_{t_y = 0}^{D - t_i - t_x} f_{i+1,j}(t_y)) \tag{4.5}$$

Now the probability we want to predict has been expressed in the form of pdf of inter-arrival times. Our delay models capture the distribution of all inter-arrival times that we need for computing the above probability: we use the delay model for the first response when $i = 0$, and use the inter-arrival of adjacent response when $i > 0$. Therefore, we can predict the delay of receiving any remaining responses given the time that partial responses are received.

4.5.4 Predicting Validation Results

Having presented the delay model, we discuss how to predict the actual content of the incoming responses, *i.e.* whether each response is a Yes or No. Specifically, given that we have received a sequence S_i, we want to compute the probability of occurrence of each possible sequence S_j that starts with S_i, such that the validation result is positive, *i.e.*, $majority(S_j) = Yes$.

This prediction can be easily done using a sufficiently large training dataset to study the distribution of all possible result sequences. For the case where the number of duplicate is set to be 5, there are $2^5 = 32$ different sequence combinations. We can compute the probability that each sequence occurs in the training set by counting the number of their occurrences. We use this probability distribution as our model for predicting validation results. We use the probabilities to construct a probability tree called *SeqTree*.

Figure 4.5 shows an example of a SeqTree tree. It is a binary tree where leaf nodes are the sequences with length of 5. For two leaf nodes where only the last bit is different, they have a common parent node whose sequence is the common substring of the two leaf nodes. For example, nodes 'YNYNN' and 'YNYNY' have a parent node of 'YNYN'. The probability of a parent node is the summation of the probability from its children. Following this rule, the SeqTree is built, where each node S_i is associated with a probability p_i that its sequence can happen.

Given the tree, it is easy to predict the probability that S_j occurs given partial sequence S_i using the SeqTree. Simply find the nodes that correspond to S_i and S_j respectively, and the probability we want is p_j/p_i.

4.6 System Implementation

The CrowdSearch system is implemented on Apple iPhones and a backend server at UMass. The components diagram of CrowdSearch system is shown in Figure 4.6.

iPhone Client: We designed a simple user interface for mobile users to capture query images and issue a search query. The screenshot of the user interface is shown in Figure 4.1. A client can provide an Amazon payments account to facilitate the

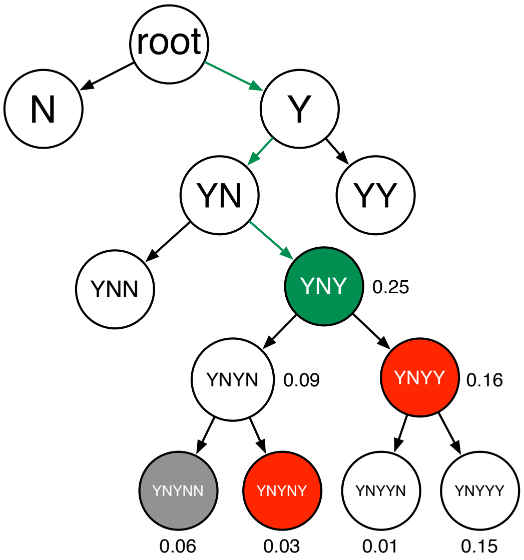


Figure 4.5. A SeqTree to Predict Validation Results. The received sequence is 'YNY', the two sequences that lead to positive results are 'YNYNY' and 'YNYY'. The probability that 'YNYY' occurs given receiving 'YNY' is 0.16/0.25 = 64%

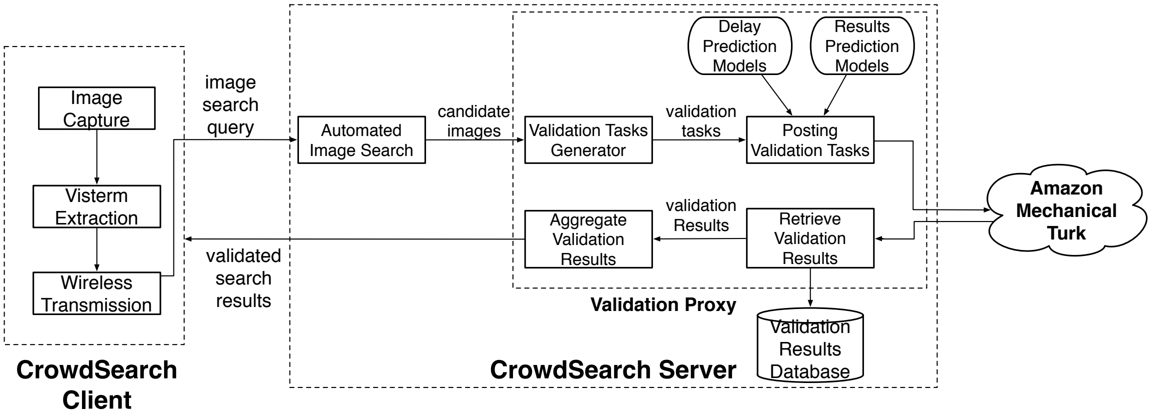


Figure 4.6. CrowdSearch Implementation Components Diagram

use of AMT and pay for validation. There is also a free mode where validation is not performed and only the image search engine results are provided to the user.

To support local image processing on the iPhone, we ported an open-source implementation of the SIFT feature extraction algorithm [117] to the iPhone. We also implemented a vocabulary tree lookup algorithm to convert from SIFT features to visterms. While vocabulary tree lookup is fast and takes less than five seconds, SIFT takes several minutes to process a VGA image due to the lack of floating point support on the iPhone. To reduce the SIFT running time, we tune SIFT parameters to produce fewer SIFT features from an image. This modification comes at the cost of reduced accuracy for image search but reduces SIFT running time on the phone to less than 30 seconds. Thus, the overall computation time on the iPhone client is roughly 35 seconds.

When the client is connected to the server, it also receives updates, such as an updated vocabulary tree or new deadline recommendations.

CrowdSearch Server Implementation: The CrowdSearch Server is comprised of two major components: automated image search engine and validation proxy. The image search engine generates a ranked list of candidate images for each search query from iPhone clients. Our current implementation of the image search engine is about 5000 lines of C++ code, and we use hierarchical k-means clustering algorithms [87] to build this image search engine. Over 1000 images collected from several vision datasets [87, 84] are used as the training images, which cover four categories: human faces, flowers, buildings, and book covers.

The validation proxy is implemented with approximately 10,000 lines of Java and Python code. There are four major components in validation proxy as shown in Figure 4.6. The Validation Task Generator module converts each <query image, candidate image> pair to a human validation task. The Task Posting module posts validation tasks to AMT using the Amazon Web Service(AWS) API. The results are retrieved by the Retriever module, which also performs some bookkeeping such as approval of tasks, and payment to the AMT solvers. These operations are implemented with AWS API as well. The Result aggregation module generates an appropriate responses from validators' answers by using a majority rule, and returns the final validated search results back to the mobile user.

4.7 Experimental Evaluation

We evaluate CrowdSearch over several thousand images, many tens of days of traces on the Amazon Mechanical Turk, and an end-to-end evaluation with a complete system running on mobile phones and a backend server. We evaluate four aspects of the performance of CrowdSearch : (a) improvement in image search precision, (b) accuracy of the delay models, (c) ability to tradeoff cost and delay, and (d) end-to-end system performance.

4.7.1 Datasets

Three datasets are used in our study.

Image Dataset: Four categories of images, including human faces, flowers, buildings, and book covers, are chosen as the dataset in our study. These four categories of images cover the precision spectrum of image search engines, from the extremely poor end to the extremely good end. Each dataset contains two parts: 1) training images that are used to build the image search engine, and 2) 500 query images captured by cellphone cameras to evaluate the performance of the search engine, and to generate human validation tasks to AMT. Note that query images are completely different from training images.

Training Dataset: From query images for each category, we randomly choose 300 images, and package each query image with its candidate results as validation tasks to AMT. We use only the top-5 candidate images for each query image. Each validation task is a <query image, candidate image> pair. In other words, we have $300 \times 5 = 1500$ tasks for each category of images. The queries are posted as a Poisson process with average interval of five minutes. The task posting lasts for several days to cover all time points in a day.

The validation results and the delay traces obtained from these queries comprise our training dataset. We use this training dataset to obtain parameters for our delay and result prediction models, and fix these parameters for all results in this section.

Testing Dataset: To validate our algorithms, we choose 200 query images from each of the four categories, and package each query images with its candidate images as validation tasks to AMT. For each query image, 5 candidate images are generated by image search engine as well. The posting process is exactly the same as the training dataset. The testing dataset is obtained two weeks after the training dataset to avoid overlap in the validators and to thoroughly evaluate our models.

The testing dataset provides us with a large set of validation tasks and the delay for each of the five responses for the tasks. This allows us to compare different task posting schemes by just changing the time at which a task is posted. The delays for responses to a particular task is assumed to be the same irrespective of when the task is posted. This assumption is reasonable since validators are independent of each other, and we find that most responses are from different validators.

4.7.2 Improving Search Precision

Our first series of experiments evaluate the precision of automated image search, and how much human validation can improve this precision. *Precision* is defined as the ratio of the number of correct results to the total number of results returned to the user. All four image categories are evaluated in this experiment.

We first look at the precision of automated search shown in Figure 4.7. On the x-axis is the length of the ranked list obtained from the search engine. The result shows that the top-ranked response has about 80% precision for categories such as buildings and books but has very poor precision for faces and flowers. The precision drops significantly as the length of the ranked list grows, indicating that even top-ranked images suffers from high error rate. Therefore, we cannot present the results directly to users.

We now evaluate how much human validation can improve image search precision. Figure 4.8 plots four different schemes for human validation: first-response, majority(3), majority(5), and one-veto (i.e. complete agreement among validators). In each of these cases, the human-validated search scheme returns only the candidate images on the ranked list that are deemed to be correct. Automated image search simply returns the top five images on the ranked list.

The results reveal two key observations: First, considerable improvement in precision is irrespective of which strategy is used. All four validation schemes are considerably better than automated search. For face images, even using a single human validation improves precision by 3 times whereas the use of a majority(5) scheme improves precision by 5 times. Even for book cover images, majority(5) still improves precision by 30%. In fact, the precision using human validators is also considerably better than the top-ranked response from the automatic search engine. Second, among the four schemes, human validation with majority(5) is easily the best performer and consistently provides accuracy greater than 95% for all image categories. Majority(3) is a close second, but its precision on face and building images is less than 95%. The one-veto scheme also cannot reach 95% precision for face, flower and building images. Using the first response gives the worst precision as it is affected most by human bias and error. Based on the above observation, we conclude that for mobile users who care about search precision, majority(5) is the best validation scheme.

4.7.3 Accuracy of Delay Models

The inter-arrival time models are central to the CrowdSearch algorithm. We obtain the parameters of the delay models using the training dataset, and validate

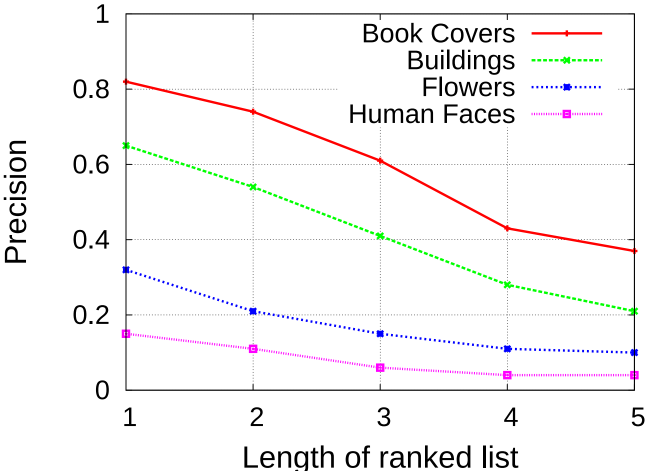


Figure 4.7. Precision of automated image search over four categories of images. These four categories cover the spectrum of the precision of automated search.

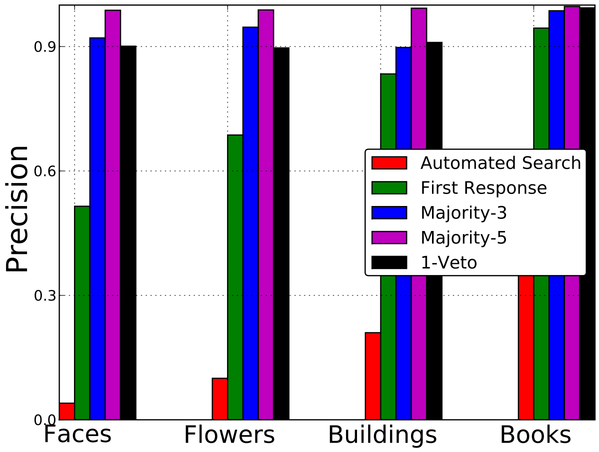


Figure 4.8. Precision of automated image search and human validation with four different validation criteria.

the parameters against the testing dataset. Both datasets are described in §4.7.1. We validate the following five models: arrival of the first response, and inter-arrival times between two adjacent responses from 1^{st} to 2^{nd} response, to 4^{th} to 5^{th} response (§4.5.3.1). In this and the following experiments, we set the threshold to post next task be 0.6. In other words, if the probability that at least one of existing validation task is successful is less than 0.6, a new task is triggered.

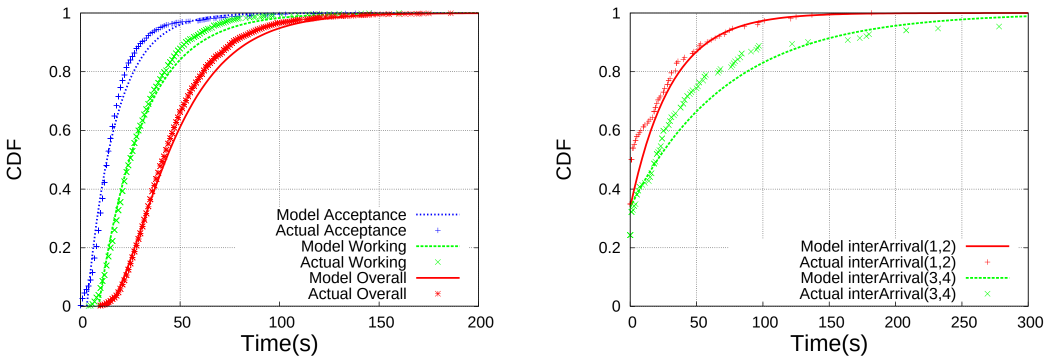
Table 4.1. KL-divergence between delay models and distribution of testing dataset

Delay	KL-Divergence
Delay for the first response	-3.44
Inter-arrival Delay for response(1,2)	-4.60
Inter-arrival Delay for response(2,3)	-1.44
Inter-arrival Delay for response(3,4)	-3.12
Inter-arrival Delay for response(4,5)	-1.81

Figure 4.9(a) shows the cumulative distribution functions (CDF) for the first response. As described in §4.5.3.1, this model is derived by the convolution of the acceptance time and submission time distribution. We show that the model parameters for the acceptance, submission, as well as the total delay for the first response fit the testing data very well. Figure 4.9(b) shows the CDF of two of inter-arrival times between 1^{st} and 2^{nd} responses, and 3^{rd} and 4^{th} responses. (The other inter-arrival times are not shown to avoid clutter.) The scatter points are for testing dataset and the solid line curves are for our model. Again, the model fits the testing data very well.

While the results were shown visually in Figure 4.9, we quantify the error between the actual and predicted distributions using the K-L Divergence metric in Table 4.1. The K-L Divergence or relative entropy measures the distance between two distributions [16] in bits. Table 4.1 shows the distance between our model to the actual data is less than 5 bits for all the models, which is very small. These values are all negative, which indicates that the predicted delay of our model is little bit larger than the actual delay. This observation indicates that our models are conservative in the sense that they would rather post more tasks than miss the deadline requirement.

The results from Figure 4.9 show that the model parameters remain stable over time and can be used for prediction. In addition, it shows that our model provides an excellent approximation of the user behavior on a large-scale crowdsourcing system such as AMT.



(a) Overall delay models vs. actual values (b) Inter-arrival delay model vs. actual values

Figure 4.9. The inter-arrival delay and overall delay of predicting values and actual values. The rate for the models of overall, acceptance, working, inter-arrival(1,2) and inter-arrival(3,4) are: 0.008, 0.02, 0.015, 0.012 and 0.007.

4.7.4 CrowdSearch Performance

In this section, we evaluate the CrowdSearch algorithm on its ability to meet a user-specified deadline while maximizing accuracy and minimizing overall monetary cost. We compare the performance of CrowdSearch against two schemes: parallel posting and serial posting, described in §4.5.1. Parallel posting posts all five candidate results at the same time, whereas Serial posting processes one candidate result at a time and returns the first successfully validated answer.

We evaluate three aspects: precision, recall, and cost. 1) *Precision* is defined as the ratio of the number of correct results to the total number of results returned to the user. 2) *Recall* is defined as the ratio of number of correctly retrieved results and the number of results that actually correct. 3) *Cost* is measured in dollars.

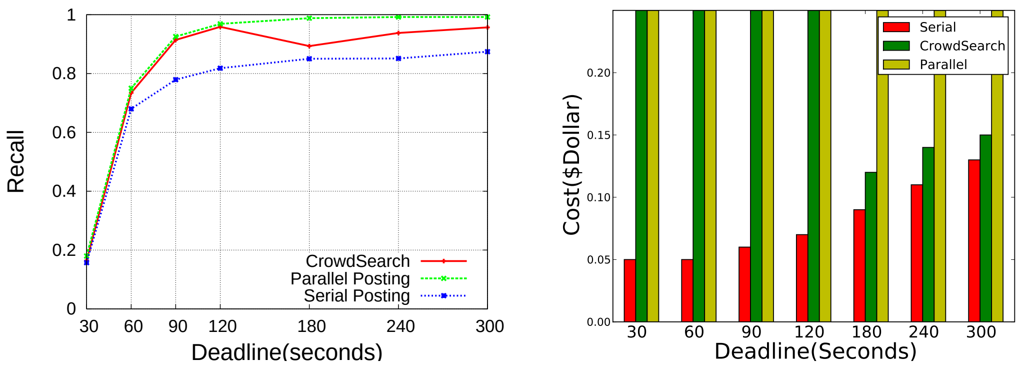
Varying user-specified deadline: To understand how the CrowdSearch algorithm works for different user-specified deadlines, we vary the deadline setting for the algorithm, and evaluate the precision, recall and cost performance of CrowdSearch. We study the testing trace for the buildings dataset. When the deadline is too low to receive a majority(5) response, all three algorithms (serial posting, parallel posting, and CrowdSearch) returns a response based on the majority of the received results.

Thus, these schemes can provide an answer even if only one or three responses are received for a task.

Our experiments show that the precision for all three schemes are higher than 95% irrespective to the deadline setting. This is because human validation is intrinsically very good and has high accuracy. While precision is high, the impact of a short deadline can be observed in the recall parameter. Short deadlines may be insufficient to obtain a positive response, leading to the three posting schemes missing valid candidate images from the search engine.

Figure 4.10(a) shows the recall as a function of the user-specified delay. We first consider recall for the serial and parallel posting schemes. At extremely low deadline (30 sec), neither scheme obtains many responses from human validators, hence the recall is very poor and many valid images are missed. The parallel scheme is quicker to recover as the deadline increases, and recall quickly increases to acceptable regions of more than 90% at 90 seconds. The serial scheme does not have enough time to post multiple candidates, however, and is slower to improve. The performance of CrowdSearch is interesting. For stringent deadlines of 120 seconds or lower, CrowdSearch posts tasks aggressively since its prediction correctly estimates that it cannot meet the deadline without posting more tasks. Thus, the recall follows parallel posting. Beyond 120 seconds, CrowdSearch tends to wait longer and post fewer tasks since it has more slack. This can lead to some missed images which leads to the dip at 180 seconds. Again the recall increases after 180 seconds since the CrowdSearch has better prediction with more partial results.

Figure 4.10(b) shows the average price per validation task as a function of the user-specified delay. When the deadline is small, CrowdSearch behaves similar to parallel search, thus the cost of these two schemes are very close. When deadline is larger than 120 seconds, the cost of CrowdSearch is significantly smaller and only 6-7% more than serial search. In this case, the deadline recommendation provided by



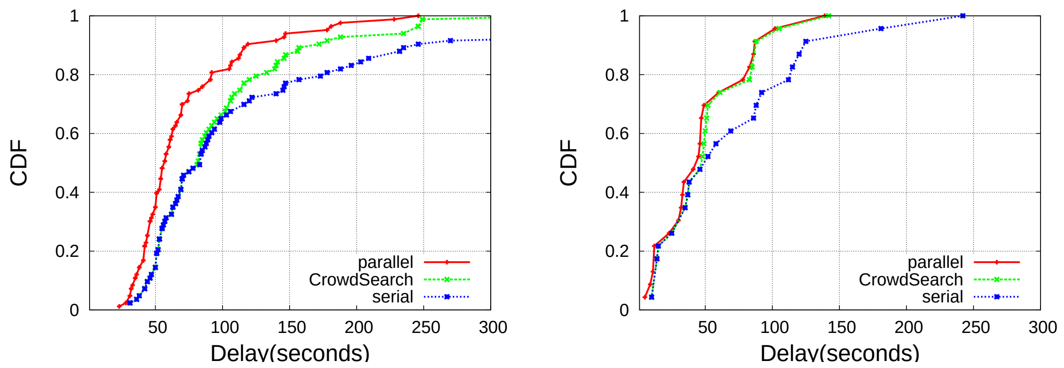
(a) Recall

(b) Monetary Cost

Figure 4.10. Recall and Cost with different deadlines using building images. The recall of Crowdsearch is close to parallel, while the cost of Crowdsearch is close to serial.

the CrowdSearch system to the user would be 180 or 240 seconds to obtain a balance between delay, cost, and accuracy in terms of precision and recall.

Breakdown for fixed deadline: To better understand how CrowdSearch improves over alternate approaches, we drill down into the case where the delay is fixed at 5 minutes (300 seconds). Figure 4.11(a) and 4.11(b) show the CDF of delay for building images and face images respectively. In both cases, we see that CrowdSearch tracks the serial posting scheme when the overall delay is low, since validation results are coming in quickly to post one task after the other. As the delay increases, CrowdSearch tracks the parallel posting scheme since it observes that the deadline is approaching and decides to post multiple tasks in-order to maximize chances of obtaining at least one valid response. Another interesting observation is about the case of faces. CrowdSearch posts more tasks after about 45 seconds, whereas it waits until about 75 seconds to become more aggressive in the case of the building dataset. This difference is because automated image search is more accurate for buildings, hence CrowdSearch receives more positive validation responses and decides to wait longer.



(a) Delay of Building Images

(b) Delay of Face Images

Figure 4.11. Search delay of parallel posting, serial posting, and CrowdSearch on two different categories of images: buildings and faces. These two categories correspond to the best and worst performance of automated image search.

How much cost does CrowdSearch incur? Figure 4.12 compares the monetary cost of Serial posting and CrowdSearch for building images and face images respectively. In these two cases, parallel posting always incurs the highest cost of 25 cents per query, since it posts all 5 candidate images at the beginning. For building images, the monetary cost of CrowdSearch is 50% smaller than the parallel algorithm, and only 10-15% more than serial algorithm, which has the minimum possible cost. For face images, the difference between CrowdSearch and parallel algorithm is 10%, and the difference with serial posting is only 2%.

The above experiments shows that CrowdSearch has delay performance close to parallel posting, and monetary cost close to serial posting. Since parallel posting has the lowest possible delay, and serial posting has lowest possible cost, CrowdSearch achieves an excellent balance between between delay and cost.

4.7.5 Overall Performance

Finally, we evaluate the overall performance of CrowdSearch system comprising iPhones, a remote server, and crowdsourcing systems. Our evaluation covers two aspects of the CrowdSearch system: energy efficiency and end-to-end delay. In prior

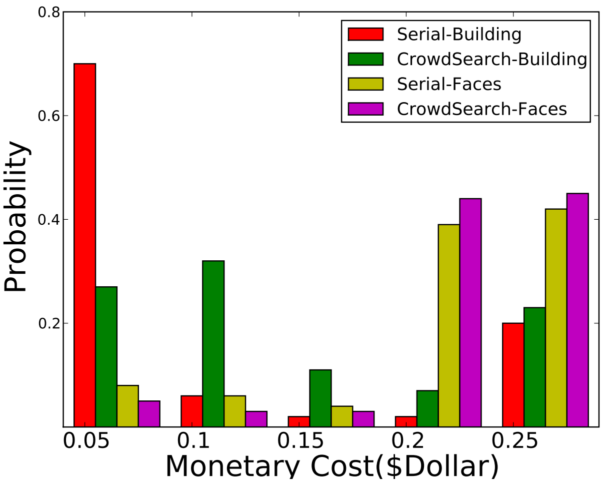
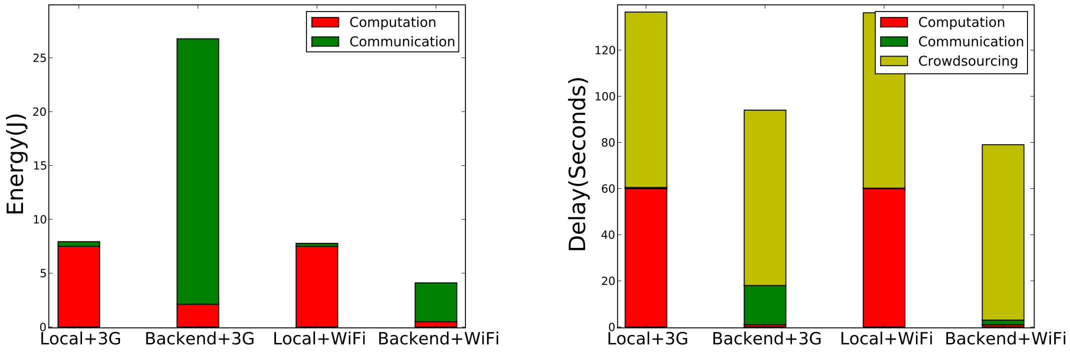


Figure 4.12. Monetary cost of parallel posting, serial posting, and CrowdSearch on two different categories of images: buildings and faces. These two categories correspond to the best and worst performance of automated image search.

work [130], we showed that tuning the parameters of SIFT processing can reduce image search accuracy by up to 30%. We do not re-evaluate this result here.

Energy Efficiency : We now provide benchmarks for the energy-efficiency of the image search engine. We consider two design choices: 1) remote processing where phones are used only as a query front-end while all search functionality is at the remote server, and 2) partitioned execution, where the visterm extraction is performed on the phone and the search by visterms is done at the remote server. In each design, we consider the case where the network connection is via AT&T 3G and WiFi. The query is a VGA resolution image.

Figure 4.13(a) shows the average energy consumption for the four combinations of design and network. The numbers represent averages over 100 runs. The results show that the energy consumption of the partitioned scheme is the same irrespective of 3G or WiFi. This is because visterms are extremely compact and do not add much to the thumbnail image that is transmitted to the server for human validation. The energy-efficiency of the remote processing scheme greatly depends on the network used. With



(a) Energy consumption

(b) End-to-end delay

Figure 4.13. Overall energy consumption and end-to-end delay for CrowdSearch system. Overall energy consumption is comprised of computation and communication. End-to-end delay is comprised of delay caused by computation, communication and crowdsourcing of AMT.

WiFi connectivity, remote processing is more efficient than local processing since transmitting a VGA image only takes a few seconds. In contrast, communicating the image via 3G is considerably more expensive, as 3G has greater power consumption and lower bandwidth than WiFi. In this case, partitioned processing can provide considerable energy gains.

Our results confirm our design choice of using remote processing when WiFi is available and local processing when only 3G is available.

End-to-end Delay: Dynamic partitioning of search functions also leads to different end-to-end delays. In this experiment, we evaluate the end-to-end delay for local and remote processing under 3G and WiFi. The crowdsourcing aspect of our system uses a 90 second deadline (although it returns results earlier if available). The results are averaged over 200 images. Figure 4.9 shows that local processing can be a significant fraction of the overall delay. While local processing consumes roughly a minute in our implementation, we believe that there are several optimizations that can be performed to reduce this delay. Human validation delay is about 4 times the delay incurred for transmitting the VGA image over a 3G network. We believe that as crowdsourcing systems increase in the number of validators that they support, the overall delay can reduce even further.

4.8 Summary

Multimedia search presents a unique challenge. Unlike text that provides sufficient context to facilitate search, search using images is difficult due to the unavailability of clear features. The explosive growth of camera-equipped phones makes it crucial to design precise image search techniques. However, despite significant research in the area, a general image search system is still far from reality. On the other hand, humans are excellent at distinguishing images, thus human validation can greatly improve the precision of image search. However, human validation costs time and money, hence we need to dynamically optimize these parameters to design an real-time and cost-effective system.

In this chapter we have presented CrowdSearch, an image search service for mobile phones where crowdsourced human validation is used to improve the search accuracy. CrowdSearch uses adaptive techniques based on accurate models of delay and accuracy behavior of human solvers. Our system is designed on iPhones, a backend server, and crowdsourcing system, such as Amazon Mechanical Turk. We demonstrate an 95% and above search precision, while being adaptive to the deadline needs for queries. Compare to alternative approaches that with similar search delay, our scheme saves the monetary cost up to 50%. While our work focuses on image search, our techniques, especially the AMT behavior models and prediction algorithms, open up a spectrum of possibilities and can be applied to areas beyond images to any multimedia search from mobile phones.

CHAPTER 5

SENSEARCH: ENERGY EFFICIENT IMAGE SEARCH FOR MOBILE PHONES

5.1 Introduction

In this chapter, we extend image search from cloud-based centralized search to distributed manner, where each smartphone performs as a micro image search engine that indexes and provide search functionality to images stored locally on phones. Specifically, we tackle the challenges on energy efficiency for distributed image search, and propose novel storage and index mechanisms to address the energy concern.

With the ubiquity of smartphones, the availability of cameras on almost all cellphones available today presents tremendous opportunities for "participatory image sensing". Mobile phone-centric applications include microblogging ([39]), telemedicine (e.g. diet documentation [92]), and others [52]. Smartphone users can also participate to sensing applications that once rely on dedicated sensor networks. The range of participatory sensing applications can include habitat monitoring [69], surveillance [42], security systems [44], monitoring old age homes[3], etc.

In these camera sensing-driven applications, smartphones are treated as a *distributed camera sensor network*, where each individual smartphone user is a human sensor that collects, stores, and transmits image sensing information to other users or to a centralized service. While smartphone camera sensing present many exciting application opportunities, how to efficiently process the sensing data is challenging for two reasons. First, the size of images is large. In contrast to sensor network data management systems for low data-rate sensors such as temperature that can continually stream data from sensors to a central data gathering site, transmitting all but a small number of images is impractical due to energy constraints. For example, transmitting a single VGA-resolution image over a low-power CC2420 wireless radio takes up to a few minutes, thereby incurring significant energy cost. Second, processing images requires sophisticated computational algorithms. For instance, "simple" computer vision algorithms such as background subtraction, or color histogram calculation, take significant amount of CPU power, and thereby incur high energy cost.

Instead of application-specific image processing technique for smartphone camera sensor networks, we have observed a trend of designing general-purpose image search paradigm that can be used to recognize a variety of objects, including new types of objects that might be detected. This can enable more flexible use of a camera sensor network across a wider range of users and applications. For example, in a habitat monitoring application, many different end-users may be able to use a single implementation of camera sensing algorithm, i.e., object recognition and search, for their diverse goals including monitoring different types of birds or animals. Two recent technology trends make a compelling case for a search-based camera sensor network. The first trend is recent advances in image representation and retrieval makes it possible to efficiently compute and store compact image representations (referred to as visual terms or visterms [33, 99]), and efficiently search through them using techniques adapted from text retrieval [86, 87]. Second, flash memory storage is cheap, plentiful and extremely energy-efficient [71], hence images can be stored locally at sensor nodes for long durations and retrieved on-demand.

5.1.1 SneSearch Overview

While distributed search in smartphone camera sensor networks (referred to as camera sensor networks, or CSN in the following chapter) opens up numerous new opportunities, it also presents many challenges. First, the resource constraints of smartphone platforms necessitate efficient approaches to image search in contrast to traditional resource-intensive techniques. Second, designing an energy-efficient image search system at each sensor necessitates optimizing local computation and local storage on flash. Finally, distributed search across such a network of camera sensors requires ranking algorithm to be consistent across multiple sensors by merging query results. In addition, there is a need for techniques to minimize the amount of communication incurred to respond to a user query.

In this chapter, we describe a novel general distributed image search architecture comprising a smartphone camera sensor network where each smartphone is a local search engine that senses, stores and searches images. The system is designed to efficiently (in terms of both computation and communication) merge scores from different local search engines to produce a unified global ranked list. Our search engine is made possible by the use of compact image representations called visterms for efficient communication and search, and the re-design of fundamental data structures for efficient flash-based storage and search. More specifically, our contributions are as follows:

- **Efficient Local Storage and Search**: The core of our SenSearch system is an image search engine at each sensor node that can efficiently search and rank matching images, and efficiently store images and indexes of them on local flash memory. Our key contributions in this work include the use of an efficient image descriptor using "visterms" (see next section) and the re-design of two fundamental data structures in an image search engine — the vocabulary tree and the inverted index — to make it efficient for flash-based storage on resource-constrained sensor platforms. We show that our techniques improve the energy consumption and response time of performing local search by 5-6x over alternate techniques.

- ▶ **Distributed Image Search**: Our second contribution is a novel distributed image search engine that unifies the local search capabilities at individual nodes into a networked search engine. Our system enables seamless merging of scores from different local search engines across different sensors to generate a unified ranked list in response to queries. The compact image description in terms of visterms (see next section) minimizes communication overhead. We show that such a distributed search engine enables a user to query a sensor network in an energy-efficient manner using an iterative procedure involving the communication of local scores, representations and images to reduce energy consumption. Our results show that such a distributed image search is up to 5x more efficient than alternate approaches, while incurring reasonable increase in latency (less than four seconds in a four-hop network).
- ▶ **Application Case Studies**: We evaluate and demonstrate the performance of our distributed image search engine in the context of an indoor library monitoring application. We show that our system achieves up to 90% accuracy for search book cover images. We also show how system parameters can be tuned to tradeoff query accuracy for energy efficiency and response time.

5.2 Related Work

In this section, we review closely related prior work on camera sensor networks, flash-based storage structures, and distributed search and image recognition.

5.2.1 Camera Sensor Networks

Much research on camera sensor networks has focused on the problem of image recognition, activity recognition (e.g.: [67]), tracking and surveillance (e.g. [44]). These systems are designed with specific applications in mind. In our design, a distributed camera search engine provides the ability for users to pose a broad set of queries thereby enabling the design of more general-purpose sensor networks. While our prototype focuses on any type of books, one can easily change this to other kinds of similar object and scenes provided appropriate features are available for that object or scene. SIFT features, are good for approximately planar surfaces like books and buildings [87] and may be even appropriate for many objects where a portion of the image is locally planar. SIFT features are robust to factors like viewpoint variation, lighting, shadows, sensor variation and sensor resolution. Further, robustness is achieved using a ranking framework in our case. There has also been work on optimizing SIFT performance in the context of sensor platforms [54]. However, their focus is on a specialized image recognition problem rather than the design of a search engine.

5.2.2 Flash-based Storage

There have been a number of recent efforts at designing flash-based storage systems and data structures including FlashDB [83], Capsule [70], ELF [27], MicroHash [132], and others. Among these, the closest are FlashDB, MicroHash and Capsule: FlashDB presents an optimized B-tree for flash, MicroHash is an optimized Hash Table for flash, and Capsule provides a library of common storage objects (stack, queue, list, array, file) for NAND flash. The similarities between our techniques and the approaches used by prior work is limited to the use of log-structured storage. Other aspects of our data structures such as the sorted and batched access of the vocabulary tree, and exploiting the Zipf distribution of the visterms are specific to our system.

5.2.3 Image Search and Object Recognition for Sensors

While there has been an enormous amount of work on image search in resource-rich server-class systems, image search on resource-constrained embedded systems has received very limited attention. The closest work is on text search in a distributed sensor network [106, 123]. However, their work assumes that users can annotate their data to generate searchable metadata. In contrast, our system is completely automated based on embedded object recognition techniques and does not require human endeavor in the loop.

5.3 Image Search Background

Before presenting the design of our system, we first provide a concise background on the state of art image search techniques, and identify the major challenges in the design of an embedded image search engine for sensor networks. Image search involves three major steps: (a) extraction of distinguishing features from images, (b) clustering features to generate compact descriptors called visterms, (c) ranking matching results for a query, based on a weighted similarity measure called tf-idf ranking [12].

5.3.1 Image Search Overview

Image Features: A necessary pre-requisite for performing image search is the availability of distinguishing image features. While such features are not available for all kinds of image types and recognition tasks, several promising techniques have emerged in recent years. In particular, when one is interested in searching for images of the same object or scene, a good representation is obtained using the Scale-Invariant Feature Transform (SIFT) [62], which generates 128 dimensional vectors by essentially computing local orientation histograms. Such a SIFT vector is typically a good description of the local region. While a number of SIFT variants like GLOH and PCA-SIFT are available, a comparison [73] shows that GLOH and SIFT work best.

Visterms or Visual Words: While search can be performed by directly comparing SIFT vectors of two images, this approach is very inefficient. SIFT vectors are continuous 128 dimensional vectors and there are several hundred SIFT vectors for a VGA image. This makes it expensive to compute a distance measure for determining similarity between images. State-of-the-art image search techniques deal with this problem by clustering image features (e.g. SIFT vectors) using an efficient clustering algorithm such as hierarchical k-means [86], and by using each cluster as a visual word or visterm, analogous to a word in text retrieval [99].

The resulting hierarchical tree structure is referred to as the *vocabulary tree* for the images, where the leaf clusters form the "vocabulary" used to represent an image [86]. The vocabulary tree contains both the hierarchical decomposition and the vectors specifying the center of each cluster. Since the number of bits needed to represent the vocabulary is far smaller than the number of bits needed to represent the SIFT vector, this representation is very efficient. We replace each 128 byte SIFT vector with a 4 byte visterm.

Matching: Image matching is done by comparing visterms between two images. If two images have a large number of visterms in common they are likely to be similar. This comparison can be done more efficiently by using a data structure called the *inverted index* or inverted file [12], which provides a mapping between a visterm and all images that contain the visterm. Once the images to compare are looked up using the inverted index, a query image and a database image can be matched using visterms by scoring them. As is common in text retrieval, scoring is done by weighting more common visterms less than rare visterms [12, 86, 87]. The rationale is that if a visterm occurs in a large number of images, it is poor at discriminating between them. The weight for each visterm is obtained by computing the tf-idf score (term frequency - inverse document frequency) as follows:

$$\begin{aligned} tf_v &= \text{Freq. of visterm v in an image} \\ df_v &= \text{Num. of images in which visterm v occurs} \\ idf_v &= \frac{\text{Total num of images}}{\mathrm{df}_v} \\ score &= \sum_i \log(tf_i + 1) \cdot \log(idf_i) \end{aligned} \tag{5.1}$$

where the index i is over visterms common to the query and database image and df_v denotes the document frequency of v. Once the matching score is computed for all images that have a visterm in common with the query image, the set of images can be ranked according to this score and presented to the user.

5.3.2 Problem Statement

There are many challenges in optimizing such an image search system for the energy, computation, and memory constraints of sensor networks. We focus on three key challenges in this chapter:

- ▶ **Flash-based Vocabulary Tree:** The vocabulary tree data structure is typically very large in size (many tens of MB) and needs to be maintained on flash. While the data structure is static and is not modified once constructed, it is heavily accessed for lookups since every conversion of an image feature to visterm requires a lookup. Thus, our first challenge is: *How can we design a lookup-optimized flash-based vocabulary tree index structure for sensor platforms?*
- ▶ **Flash-based Inverted Index:** A second important data structure for search is the inverted index. As the number of images captured by a sensor grows, the inverted index will grow to be large and needs to be maintained on flash. Unlike the vocabulary tree, the inverted file is heavily updated since an insertion operation occurs for every visterm in every image. In contrast, the number of lookups on the inverted index depends on the query frequency, and can be expected to be less frequent. Thus, our second challenge is: *How can we design an update-optimized flash-based inverted index for sensor platforms?*
- ▶ **Distributed Search:** Existing image search engines are designed under the assumption that all data is available centrally. In a sensor network, each node has a unique and different local image database, therefore, we need to address

questions about merging results from multiple nodes. In addition, sensor network users can pose different types of queries — continuous and ad-hoc — which need to be efficiently processed. Thus, our third challenge is: *How can we perform efficient and accurate distributed search across diverse types of user queries and diverse local sensor databases?*

The following sections discusses our overall architecture followed by techniques employed by our system to address the specific problems that we outlined in this section.

5.4 Image Search Architecture

We now provide a broad overview of the operation of our distributed image search system, which we describe in the context of a bird monitoring sensor network. We assume that users of such a sensor network wish to pose archival queries on stored images as well as continuous queries on live images. An example of an archival query might be to retrieve the top five matches of a user-provided query image, say a hawk, that were detected by the sensor network in the last month. The user can also pose a continuous query, and request to be notified whenever a newly captured image matches the query image of the hawk. Such image search over a sensor network involves the following steps:

Image Capture and Feature Extraction: The first step involves capture of images by a camera sensor node, perhaps in a periodic or triggered manner. A simple motion detection algorithm may be used to filter images such that only potentially interesting images are retained for further processing. Once such an image is captured, each sensor extracts descriptive features using the SIFT algorithm and maps these features to visterms as described in Section 5.3. This process of mapping SIFT features to visterms involves looking up a vocabulary tree which can either be preloaded onto the sensor during deployment time, or dynamically downloaded from a server during in-situ operation.

Local Search: Search can proceed in two ways. One approach ("Query by Visterms") is to do local search at sensors *i.e.* to transmit visterms of query images to individual sensors, perform the search locally at the sensor nodes, and merge results from multiple sensors to generate a global search result. An alternate approach ("Collect Visterms") is to do *centralized search i.e.* to push visterms for all captured images from all sensors to a central proxy which indexes these visterms to search across images stored at sensors. While both modes of operation are supported by our system, their relative efficiencies depend on specific sensor platforms and infrastructure availability. Here, we restrict our discussion to the local search mechanism since it is a more general paradigm.

The local image search engine handles continuous and archival queries in different ways. For an archival query, the visterms of the image of interest to the query are matched against the visterms of all the images that are stored in the local image store. The result of the local search is a top-k ranked list of best matches, which is transmitted to the proxy. For a continuous query, each captured image is matched against the specific images of interest to the query, and if the match is greater than a pre-defined threshold, the captured image is considered a match and transmitted to the proxy.

Global Search: Once the local search engine has searched for images matching the query, the proxy and the sensor interact to enable global search across a distributed sensor network. This interaction is shown in Figure 5.1. Global search involves combining results from multiple local search engines at different sensors to ensure that communication overhead is minimized across the network. For instance, it is wasteful if each sensor transmits images corresponding to its local top-k matches since the local top-k matches may not be the same as the global top-k matches to a

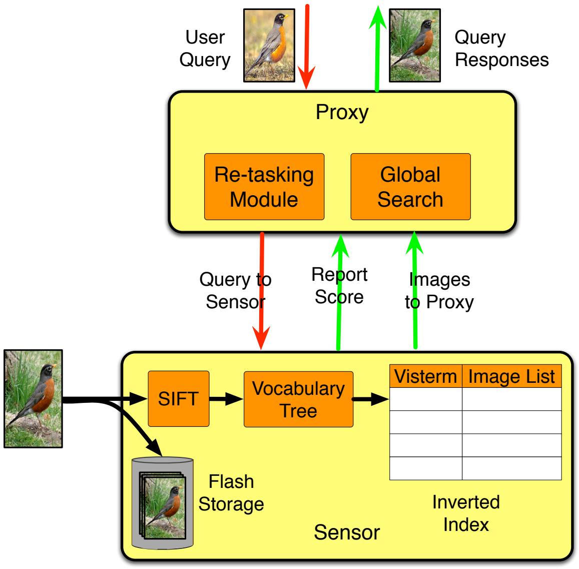


Figure 5.1. Search Engine Architecture

query. Instead, the proxy gets the ranking scores corresponding to the top-k matches resulting from the local search procedure at each sensor. The proxy merges the scores obtained from different sensors to generate a global top-k list of images, which it communicates to the appropriate sensors. The sensors then transmit thumbnails or the full images of the requested images, which are presented to the user using a GUI.

5.5 Buffered Vocabulary Tree

The vocabulary tree is the data structure used at each sensor to map image features (for example SIFT vectors) to visual terms or visterms. We now provide an overview of the operation of the vocabulary tree, and describe the design of a novel index structure, the Buffered Vocabulary Tree index structure, that is optimized for flash-based lookups.

5.5.1 Description

The vocabulary tree at each sensor is used to map SIFT vectors extracted from captured images to their corresponding visterms that are used for search. The vocabulary tree is typically created using a hierarchical k-means clustering of all the SIFT features in the image database [86]. Hierarchical k-means assumes that the data is first clustered into a small number of m clusters. Then at the next level, the points in each of the m clusters are clustered into a further m clusters so that this level has m^2 clusters. The process is repeated so that at a depth d the number of clusters is m^d. The process stops when the desired number of leaf clusters is reached. For example with $m = 10$ and $d = 6$ there are a million clusters at the leaf level. The clusters at the leaf level correspond to visual words or visterms. The ID of the leaf node is the ID of the visterm; for example, if a SIFT vector is mapped to the second cluster in the vocabulary tree, its visterm ID is 2. In the resulting vocabulary tree, each level has a number of clusters, where each cluster is represented by the coordinates of the cluster center, as well as pointers from each node to its parent, children, and siblings.

Lookup of the tree proceeds in a hierarchical manner where the SIFT vector is first compared with the m cluster centers at the top level and assigned to the cluster with the closest center at this level c_k. Then, the SIFT vector is compared with the centers of the siblings of c_k and assigned to the closest sibling c_{kj}. The process repeats until the SIFT vector is assigned to a leaf node or visterm.

5.5.2 Design Goals

The design of the vocabulary tree has two key objectives:

- *Minimize tree construction cost:* The process of constructing the vocabulary tree for a set of database images is computationally intensive. Thus, our first goal is to minimize the cost of tree construction.

- *Minimize Flash reads:* The vocabulary tree is a large data structure (few to many MB), hence it may not be possible to load the data structure completely to memory on a memory-constrained embedded platform (e.g. iMote2). Alternatively, we need to store it on flash and load it partially for visterm lookup.

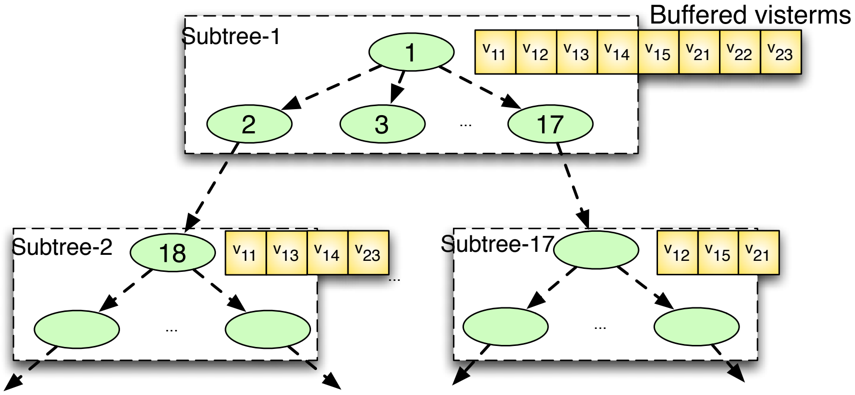

Buffered visterms
Subtree-1
1
2
3
...
17
v_{11} v_{12} v_{13} v_{14} v_{15} v_{21} v_{22} v_{23}
Subtree-2
18
v_{11} v_{13} v_{14} v_{23}
...
...
Subtree-17
v_{12} v_{15} v_{21}
...

Figure 5.2. Vocabulary Tree

Since reading the vocabulary tree from flash incurs considerable latency, and consequently energy, our second goal is to minimize the number of reads from flash for every lookup of the vocabulary tree.

5.5.3 Proxy-based Tree Construction

Unlike conventional search engines where the vocabulary tree is created from all the images that need to be searched, our approach separates the images used for tree *construction* from those used for *search*. The proxy constructs the vocabulary tree from images similar (but not necessarily identical) to those we expect to capture within the sensor network. For example, in a book search application, a training set can comprise a variety of images of book covers for generating the vocabulary tree at the proxy. The images can even be captured using a different camera with different resolution from the deployed nodes. Once constructed, the vocabulary tree can either be pre-loaded onto each sensor node prior to deployment (this can be done by physically plugging in the flash drive to the proxy and then copying it), or can be transmitted to the network during system operation.

One important consideration in constructing the vocabulary tree for sensor platforms is ensuring that its size is minimal. Previous work in the literature has shown that using larger vocabulary trees produces better results [86, 87]. This work is based on using trees with a million leaves or more which is many Gigabytes in size. Such a large vocabulary tree presents three problems in a sensor network context: (a) they are large and consume a significant fraction of the local flash storage capacity, (b) they incur greater access time to read, thereby greatly increasing the latency and energy consumption for search, and (c) they would be far too large to dynamically communicate to update sensor nodes. To reduce the size of the vocabulary tree, our design relies on the fact that typical sensor networks have a small number of entities of interest (e.g.: a few books, birds, or animals), hence, the vocabulary tree can be smaller and targeted towards searching for these entities. For example, in a book monitoring application, a few hundreds of books can be used as the training set to construct the vocabulary tree, thereby reducing the number of visterms and consequently the size of the tree.

5.5.4 Buffered Lookup

The key challenge in mapping a SIFT vector to a visterm on a sensor node is minimizing the overhead of reading the vocabulary tree from flash. A naive approach that reads the vocabulary tree from flash as and when required is extremely inefficient since it needs to read a large chunk of the vocabulary tree for practically every single lookup.

The main idea in our approach is to reduce the overhead of reads by performing batched reads of the vocabulary tree. Since the vocabulary tree is too large to be read into memory as a whole, it is split into smaller sub-trees as shown in Figure 5.2, and each subtree is stored as a separate file on flash. The subtree size is chosen such that it fits within the available memory in the system. Therefore, the entire subtree file can be read into memory. Second, we batch the SIFT vectors from a sequence of captured images into a large in-memory SIFT buffer. Once the SIFT buffer is full, the entire buffer is looked up using the vocabulary tree one level at a time. Thus, the root subtree (subtree 1 in Figure 5.2) is first read from flash, and the entire batch of SIFT vectors is looked up on the root subtree. This lookup determines which next level subtree needs to be read for each SIFT vector, and results in a smaller set of second-level buffers. The next level sub-trees are looked up one by one, and the batch processed on each of these sub-trees to generate third-level buffers, and so on. The process proceeds in a level by level manner, with a subtree file being read, and a buffer of SIFT vectors being looked up at each level. Such a buffered lookup on a segmented vocabulary tree ensures that the cost of reading an entire vocabulary tree is amortized over a batch of vectors, thereby reducing the amortized cost per lookup.

5.6 Inverted Index

While the vocabulary tree is used to determine how to map from SIFT feature vectors to visterms, an inverted index (also known as the inverted file) as in text retrieval [12] is used to map a visterm to the set of images in the local database that contain the visterm. The inverted index is used in two situations in our system. First, when an image is inserted into the local database the visterms for the image are inserted into the inverted index; this enables search by visterms, wherein the visterms contained in the query image can be matched to the visterms contained in the stored locally captured images. Second, the inverted index is also used to determine which images to age when flash is filled up to make room for new images. Aging of images proceeds by first determining the images that are least likely to be useful for future search, and deleting them from flash.

5.6.1 Description

The inverted index is updated for every image that is stored in the database. Let the sequence of visterms contained in image I_i be $\mathbf{V_i} = v_1, v_2, ..., v_k$. Then, the entry

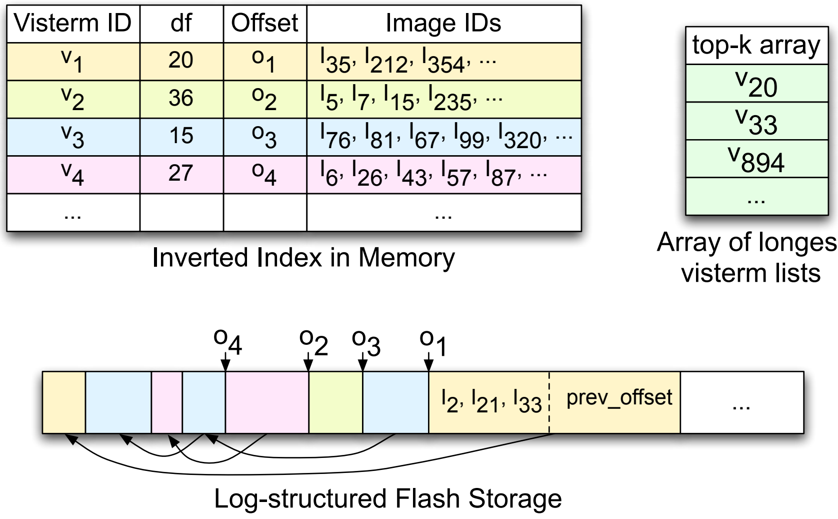
I_i is inserted into the inverted index entry for each of these visterms contained in $\mathbf{V_i}$. Figure 5.3 shows an example of an inverted index that provides a mapping from the visterm to the set of images in the local database that contain the term, as well as the frequency of the appearance of the term across all images in the local database. Each entry is indexed by the Visterm ID, and contains the document frequency (df) score of the visterm (Equation 5.1), and a list of Image IDs. We use a modified version of the scoring function 5.2 which does not use the term frequency (tf) but uses only the inverse document frequency (idf). As we show later, the idf scores are sufficient for good performance in our system. Hence, we do not store the term frequency (tf) numbers per image - this also saves valuable memory space. Since the df and idf have to be updated when new images are added, it is more efficient to store the df and compute the idf at query time.

The inverted index facilitates querying by visterms. Let the set of visterms in the query image be $\mathbf{Q} = q_1, q_2, ..., q_n$. Each of these visterms is then used to look up the inverted index and the corresponding inverted list for the visterm returned. Thus for query visterm q_i, a list of image IDs $\mathbf{L_i} = I_{i1}, ..., I_{ik}$ is returned, where each element of the list is an image ID. The lists over all the query visterms are intersected and scored to obtain a rank ordering of the images with respect to the query.

5.6.2 Design Goals

Unlike the vocabulary tree which is a static data structure that is never updated, the inverted index is updated for every visterm in a captured image, hence it needs to be optimized for insertions. The design of the flash-based inverted index structure is influenced by the following characteristics of the flash layer below and from the search engine layer above, and has the following goals:

- *Minimize flash overwrites:* Flash writes are immutable and one-time—once written, a data page must be erased before it can be written again. The smallest



Visterm ID	df	Offset	Image IDs
v_1	20	o_1	$I_{35}, I_{212}, I_{354}, \ldots$
v_2	36	o_2	$I_5, I_7, I_{15}, I_{235}, \ldots$
v_3	15	o_3	$I_{76}, I_{81}, I_{67}, I_{99}, I_{320}, \ldots$
v_4	27	o_4	$I_6, I_{26}, I_{43}, I_{57}, I_{87}, \ldots$
...			...

top-k array
v_{20}
v_{33}
v_{894}
...

Figure 5.3. Inverted Index. DF denotes document frequency

unit that can be erased on flash, termed an *erase-block*, typically spans few tens of pages, which makes any in-place overwrite of a page extremely expensive since it incurs the cost of a block read, write and erase. Hence, it is important to minimize the number of overwrites to a location that has been previously written to on flash.

- *Exploit visterm frequency:* The frequency of words in documents typically follows a heavy tailed behavior, referred to as a Zipf distribution, *i.e.* the frequency of a word is inversely proportional to its rank [12]. From a search perspective, the least frequent words are the most useful for discriminating between images, and have the highest idf score. Our third goal is to exploit this search engine characteristic to optimize the inverted index design.

5.6.3 Inverted Index Design

We discuss three aspects of the design of the inverted index in this section: (a) how the index is stored on flash, (b) how it is updated when a new image is captured, and (c) how entries are deleted when images are removed from flash.

Storage: The inverted index is maintained on flash in a *log-structured* manner, *i.e.* instead of updating a previously written location, data is continually stored as an append-only log. Within this log, the sequence of image IDs corresponding to each visterm is stored as a reverse linked list of chunks, where each chunk has a set of image IDs and a pointer to the previous chunk. For example, in Figure 5.3, the set of Image IDs corresponding to visterm v_1 is stored in flash as two chunks, with a pointer from the second chunk to the first chunk. The main benefit of this approach is that each write operation becomes significantly cheaper since it only involves an append operation on flash, and avoids expensive out-of-place rewrites. In addition, the writes of multiple chunks can be coalesced to reduce the number of writes to flash, thereby reducing the fixed cost of accessing flash for each write. Such a reverse linked list approach to storing files is also employed in a number of other flash-based data storage systems that have been designed for sensor platforms including MicroSearch [106], ELF [27], Capsule [70], and FlashDB [83].

Insertion: While a log-structured storage optimizes the write overhead of maintaining an inverted index, it increases the read overhead for each query since accessing the sequence for each visterm involves multiple reads in the file. To minimize read overhead, we exploit the Zipfian nature of the idf distribution in two ways during insertion of images to the index. First, we exploit the observation that the most frequent terms have a very low contribution to the idf score, and have least impact on search. Hence, these visual terms can be ignored and do not have to be updated for each image. In our system, we determine these non-discriminating visual terms during vocabulary tree-construction time, so that they do not need to be updated during system operation.

Second, we minimize the read overhead by only flushing the "longest chains" to flash i.e. the visterms that have the longest sequence of image IDs. Due to the Zipfian distribution of term frequency, a few chains are extremely long, and by writing these to flash, the number of flash read operations can be tremendously reduced. The Zipfian distribution also reduces the amount of state that needs to be maintained for determining the longest chains. We store a small top-k list of the longest chains as shown in Figure 5.3. This list is updated opportunistically — whenever a visterm is accessed that has a list longer than one of the entries in the top-k list, it is inserted into the list. When the size of the inverted index in memory exceeds the maximum threshold, the top-k list is used to determine which visterm sequences to flush to flash. Since the longer sequences are more frequently accessed, the top-k list contains the longest sequences with high probability, hence this technique writes the longest chains to flash, thereby minimizing the number of flash writes. The use of the frequency distribution of visterms to determine which entries to flush to flash is a key distinction between the inverted index that we use and the one that is proposed in [106].

Deletion: Image deletions or aging of images triggers deletion operations on the inverted index. Deletion of elements is expensive since it requires re-writing the entire visterm chain. To avoid this cost, we use a large image ID space (32 bits) such that rollover of the ID is practically impossible during the typical lifetime of a sensor node.

5.6.4 Using Inverted Index for Aging

In addition to search, our system also employs the inverted index is to determine what images should be aged when flash is filled up. Ideally, images that are less likely to contain objects of interest should be aged before images that are more likely to contain objects of interest. We use a combination of value-based and time-based aging of images. The “value” of an image is computed as the cumulative idf of all the visterms in the image normalized by the number of visterms in the image. Images with lower idf scores are more likely to be the background rather than objects of interest. This value measure can be combined with the time when an image was captured to determine a joint metric for aging. For example, images that are more than a week old, and have normalized idf score less than a pre-defined threshold can be selected for aging.

5.7 Distributed Search

A distributed search capability is essential to be able to obtain information that may be distributed over multiple nodes. In our system, the local search engines at individual sensors are networked into a single distributed search engine that is responsible for searching and ranking images in response to a query. In this section, we discuss how global ranking of images is performed over a sensor network.

5.7.1 Search Initiation

A user can initiate a search query by connecting to the proxy, and transmitting a search request. Two types of search queries are supported by our system: ad-hoc search and continuous search. An ad-hoc search query (also known as a snapshot query) is a one-time query, where the user provides an image of interest and, perhaps a time period of interest, and initiates a search for all images captured during the specified time period that match the image. For example, in the case of a book monitoring sensor network, a user who is missing his or her copy of “TCP Illustrated” may issue an ad-hoc query together with a cover of the missing book, and request all images matching the book detected over the past few days. A continuous search query is one where the network is continually processing captured images to decide whether it matches a specific type of entity. For instance, in the above book example, the user can also issue a continuous query and request to be notified whenever the book is observed in the network over the next week.

In both cases, the proxy which receives the query image converts the image into its corresponding visterms. The visterm representation of a query is considerably smaller than the original image (approx 1600 bytes), hence, this makes the query considerably smaller to communicate over the sensor network. The proxy reliably transmits the query image visterms to the entire network using a reliable flooding protocol.

5.7.2 Local Ranking of Search Results

Once the query image visterms are received at each sensor, the sensors initiate the local search procedure. We first describe the process by which images are scored and ranked, and then describe how this technique can be used for answering ad-hoc and continuous queries.

Scoring and Ranking: The local search procedure involves two steps: (a) the inverted index is looked up to determine the set of images to search, and (b) the similarity score is computed for each of these images to determine a ranked list. Let V_Q be the set of visterms in the query image. The first step involved in performing the local search is to find all images in the local database that have at least one of the visterms in V_Q. This set of images can be determined by looking up the inverted index for each of the entries v in V_Q. Let L_v be the list of image IDs corresponding to visterm v in the inverted index. Assume that the first visterm in V_Q is v_1 and the corresponding inverted list of images is L_{v_1}. We maintain a list of documents D, which is initialized with the list of images L_{v_1}. For each of the other visterms v in V_Q, the corresponding inverted index L_v is scanned and any image not in the document list D is added to it.

Once the set of images is identified, matching is done by comparing visterms between each image in $V(i)$, where $i \in D$ and the visterms in the query image V_Q. If the two images have a large number of visterms in common they are likely to be similar. Visterms are weighted by their *idf.* Visterms which are very common have a lower idf score, and are weighted less than uncommon visterms. The idf is computed as described in Equation 5.1.

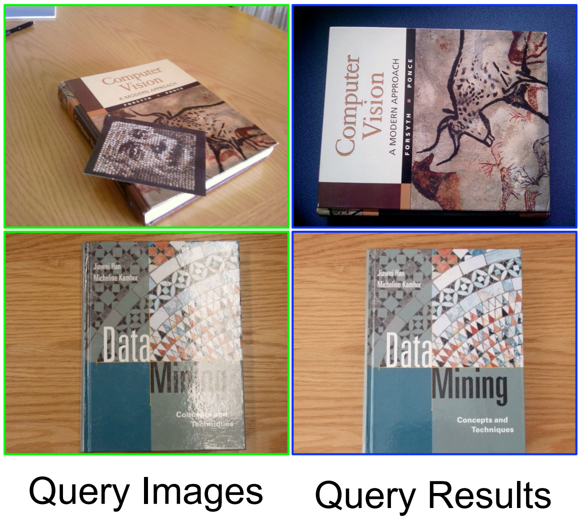


Figure 5.4. Examples of book cover search results. The first column shows queries and the top results are shown in the second column. The technique is resilient to occlusions and viewpoint change (top left image) and specularities (bottom left image).

To obtain the score of an image i, we add up the idf scores for the visterms that match between the query image, V_Q, and the database image, V_i to obtain the total score as shown in Equation 5.2. Note that the equation does not use the *tf* term from equation 5.1. This is for two reasons: (a) some non-discriminating visterms occur very frequently (e.g. background visterms), leading to false matches, and (b) the *tf* term of every visterm in every image needs to be stored, which would significantly increase the size of inverted index.

$$Score(V_i, V_Q) = \sum_{i \in V_Q and i \in V_i} \log(idf_i) \tag{5.2}$$

Any image with a score greater than a fixed pre-defined threshold ($Score(V_i, V_Q) > Th$) is considered a match, and is added to the list of query results. The final step is sorting these query results by score to generate a ranked list of results, where the higher ranked images have greater similarity to the query image.

Figure 5.4 shows example queries and the top search result for a book search example. Note the ability to handle viewpoint change, occlusion and specularities.

Ad-hoc vs Continuous Queries: The local search procedures for ad-hoc and continuous queries use the above scoring and ranking method but differ in the database images that they consider for the search. An ad-hoc query is processed over the set of images that were captured within the time period of interest to the query. To enable time-based querying, an additional index can be maintained that maps each stored image to a timestamp when it was captured. For a few thousand images, such a time index is a small table that can be maintained in memory. Once the list of images to match is retrieved from the inverted index lookup, the time index is looked up to prune the list and only considers images that were captured during the time period of interest. The rest of the scoring procedure is the same as the mechanism described above. In the case of a continuous query, the search procedure runs on each captured image as opposed to the stored images. In this case, a direct image-to-image comparison is performed between the visterms of the captured image and those of the query image. If the similarity between the visterms exceeds a pre-defined threshold, it is considered a positive match. If the number of continuous queries is high, the cost of direct matching can be further reduced by using an inverted file.

5.7.3 Global Ranking of Search Results

The search results produced by individual sensors are transmitted back to the proxy to enable global scoring of search results. The key problem in global ranking of search results is re-normalizing the scores across separate databases. As shown in Equation 5.1, the local scoring at each sensor depends on the total number of images in the local database at each sensor. Different sensors could have different number of captured images, and hence, differently sized local databases. Hence, the scores need to be normalized before comparing them.

To facilitate global ranking, each sensor node transmits the count of the number of images in which each visterm from the set V_Q occurs, in addition to the total number of images in the local database. In other words, it transmits the numerator and denominator of Equation 5.1 separately to the proxy. Note that only the numbers for the visterms which occur in the query need to be updated not all the visterms. A typical query image has about 200 visterms so we need to only send on the order of 1.6 KB from each sensor node that has a valid result. Let S be the set of sensors in the network, and C_{ij} be the count of the number of images captured by sensor s_i that contain the visterm v_j. Let N_i be the total number of images at sensor s_i. Then, the global idf of visterm v is calculated as:

$$idf_v = \frac{\sum\limits_{v_i \in S} N_i}{\sum\limits_{v_i \in S} C_{iv}} \tag{5.3}$$

Once the normalized idfs are calculated for each visterm, the scores for each image are computed in the same manner as shown in Equation 5.2. Finally, the globally ranked scores are presented to the user. The user can request either a thumbnail of an image on the list, or the full image. Retrieving the thumbnail provides a cheaper option than retrieving the full image, hence it may be used as an intermediate step to visually prune images from the list.

5.7.4 Network Re-Tasking

An important component of our distributed infrastructure is the ability to re-task sensors by loading new vocabularies. This ability to re-task is important for two reasons. First, it enables the sensor proxy to be able to update the vocabulary tree at the remote sensors to reflect improvements in the structure, for instance, when new training images are used. Second, this allows the search engine to upload smaller vocabulary trees as and when needed on to the resource constrained sensors. One of the benefits of loading smaller vocabulary trees on-demand is that it is less expensive than searching through a large vocabulary tree locally at each sensor. Third, when

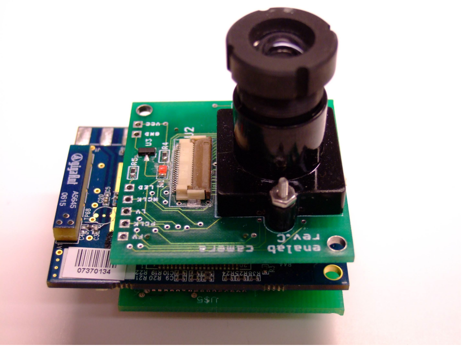


Figure 5.5. iMote2 with Enalab camera and custom SD card board

it is necessary to use the sensor network to search for new kinds of objects, a new vocabulary tree may be loaded. For example, assume we had a vocabulary tree to detect certain kinds of animals such as deer but we now want to re-task it to find tigers, we can easily build a new vocabulary tree at the proxy and download it. Dissemination of the new vocabulary into a sensor network can be done using existing re-programming tools such as Deluge [46].

5.8 Implementation

Our implementation is based on an Intel iMote2-based platform[47], with an Enalab camera [129], and a custom SD-card extension board that we designed, as shown in Figure 5.5. The Enalab camera module comprises an OV7649 Omnivision CMOS camera chip, which provides color VGA (640x480) resolution. The iMote2 comprises a Marvell PXA271 processor which runs between 13-416 MHz, and has 32MB SDRAM[47]; a Chipcon CC2420 radio chip; and an Enalab camera. The camera is connected to the Quick Capture Interface (CIF) on iMote2. To support large image data storage, an 1GB external flash memory is attached to each sensor node.

Sensor Implementation: The overall block diagram of the search engine at each sensor is shown in Figure 5.6. The entire system is about 5000 lines of C code excluding the two image processing libraries that we used for SIFT and vocabulary

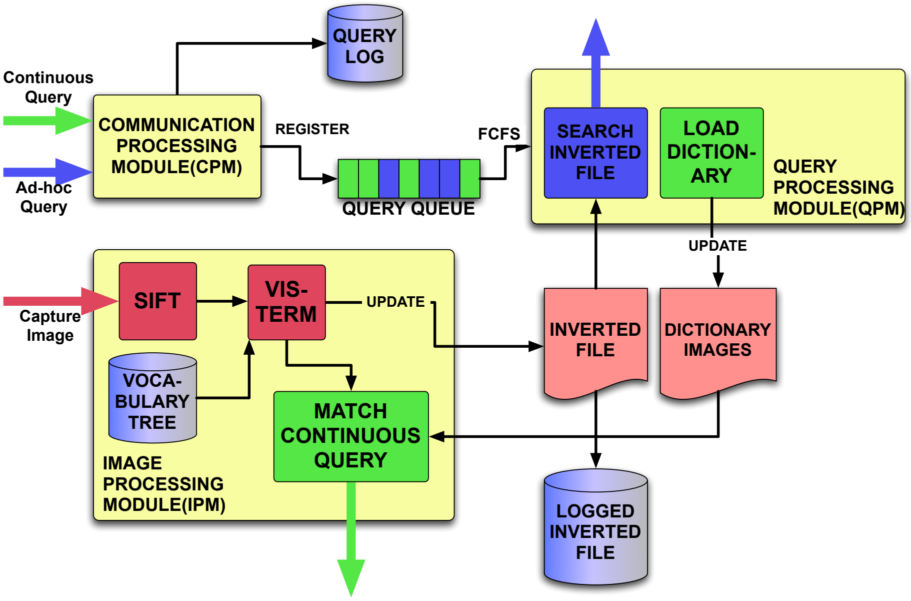


Figure 5.6. SenSearch System Architecture

tree construction. The system has three major modules: (a) Image Processing Module (IPM), (b) Query Processing Module (QPM), and (c) Communication Processing Module (CPM). The IPM captures an image using the camera periodically, reads the captured image and saves it into a PGM image file on flash. It then processes the PGM image file to extract SIFT features, and batches these features for a buffered lookup of the vocabulary tree.The QPM module processes ad-hoc queries and continuous queries from the proxy. For an ad-hoc query, it looks up the inverted file and find the best k matches from the locally stored images. If the value of k is not specified by the query, the default is set to five. For continuous queries, QPM loads the visterms of the corresponding dictionary images once a new type of query is received. Whenever a new image is captured, the IPM goes through all the dictionary images to find best k matches. CPM handles the communication to other sensors and the proxy.

SIFT Algorithm: The SIFT implementation we are using is derived from SIFT++ [98]. This implementation uses floating point for most calculations, which is exceedingly slow on the iMote2 (10 - 15 seconds to process a QVGA image) since it does not have a floating point unit. Ko et al [54] present an optimized SIFT implementation which uses fixed point arithmetic and show much faster processing speed. However, their implementation is specific to the TI Blackfin processor family. For lack of time, we have been unable to port this implementation to our node.

Vocabulary Tree: Our implementation of the hierarchical k-means algorithm to generate the vocabulary tree is derived from the libpmk library [57]. Due to memory constraints on iMote2, we set the size of the vocabulary tree to have 64K visterms, *i*.e. the branching factor = 16, and depth = 5. By modifying libpmk, we can shrink the vocabulary tree to 3.65MB, but it is still too large to be loaded completely to memory in iMote2. As described in Section 5.5, we split the entire vocabulary tree into a number of smaller segments. The root segment contains the first three levels of the tree, and for each leaf node of the root subtree, i.e., $16^2 = 256$ in our case, there is a segment containing the last two levels of the tree. After splitting, each chunk is of size 14KB, which is small enough for iMote2 to read into memory.

Inverted Index: The inverted index is implemented as a single file on flash, which is used as an append-only log file. The inverted index is maintained in memory as long as sufficient memory is available, but once it exceeds the available memory, some image lists for visterms are written to the log file. Each of the logged entries is a list of image IDs, and the file offset to the previous logged entry corresponding to the same visterm. The offset of the head of this linked list is maintained in memory in order to access this flash chain. When the image list corresponding to a visterm needs to be loaded into memory, the linked list is traversed.

Image Storage and Aging: Each raw image is stored in a separate .pgm file, its SIFT features are stored in a .key file, and its visterms are stored in a .vis file. A four-byte image ID is used to uniquely identify an image. We expect the number of image that can be captured and stored on a sensor during its lifetime to be considerably less than the 2^{32}, therefore, we do not address the rollover problem in this implementation. Aging in our system is triggered when the flash becomes more than 75% full.

Wireless Communication Layer: The wireless communication in our system is based on the TinyOS MAC driver which is built upon the IEEE 802.15.4 radio protocol. This driver provides a simple open/close/read/write API for implementing wireless communication and is compatible with the TinyOS protocols. Since there is no readily available implementation of a reliable transport protocol for the iMote2, we implemented a reliable transport layer over the CC2420 MAC layer. The protocol is simple and has a fixed window (set to 2 in our implementation) and end-to-end retransmissions. We note that the contribution of our work is not a reliable transfer protocol, hence we were not focused on maximizing the performance of this component.

5.9 Experimental Evaluation

In this section, we evaluate the efficiency of our SenSearch distributed image search system on a testbed consisting of 6 iMote2 sensor nodes, and a PC as a central proxy.

For our trace-driven experiments, we use a dataset consisting of over 600 images for our training set. Among them, over three hundred images are technical book covers, and the other three hundred images contain random objects. The book cover images are collected from a digital camera with VGA format. The other images are collected from the Internet. Note that the database images we gathered do not need any further prossessing, such as cropping or alignment, thanks to the scale invariant property of SIFT features. During our live experiments, the iMote2 captures images periodically and different technical book covers were held up in front of the iMote2 camera to form the set of captured images.

Table 5.1. Power and Energy breakdown

Component	State	Power(mW)	Per-byte Energy
PXA271 Processor	Active	192.3	
	Idle	137.0	
CC2420 radio	Active	214.9	46.39 μJ
SD Flash	Read	11.2	5.32 nJ
	Write	40.3	7.65 nJ
OV camera	Active	40.0	

5.9.1 Micro-benchmarks

Our first set of microbenchmarks are shown in Table 5.1, where we measure the power consumption of the PXA271 processor, the CC2420 radio, the SD card extension board, and the Enalab camera on our iMote2 platform. Since the data rates of different components vary, we also list the energy consumption for some of them. The results show that the processor consumes significant power on our platform, hence it is important to optimize the amount of processing. Another key observation is that the difference between the energy costs of communication and storage is significant. This is because the SD card consumes an order of magnitude less power than the CC2420 radio, and has a significantly higher data rate. The effective data rate of the CC2420 radio is roughly 46.6Kbps, whereas the data rate of the SD card is 12.5 MBps. Therefore, storing a byte is three orders of magnitude cheaper than transmitting a byte, thereby validating our extensive use of local storage.

Table 5.2 benchmarks the running time of the major image processing tasks in our system, including sensing, SIFT computation, and image compression. All of these tasks involve the processor in active mode. We find that the SIFT feature extraction from a QVGA image using the SIFT++ library [98] is prohibitively time consuming (roughly 12 seconds). A significant fraction of this overhead is because of floating point operations performed by the library, which consumed excessive time on a PXA271 since the processor does not have a floating point unit. This problem

Table 5.2. Image processing breakdown

Operation	Time(s)	Energy(J)
Sensing(Image Capture)	0.7	0.14
Sift (floating pt)	12.7	2.44
Sift(fixed pt) [54]	2.5	0.48
Compress to JPEG (ratio 0.33)	1.2	0.23
Compress to GZIP (ratio 0.61)	0.7	0.14

Table 5.3. Breakdown of memory usage

Task	Memory (MB)
Arm Linux	14.3
Image Capture	1.9
SIFT	7.6
Inverted Index	0.75
Vocabulary Tree	0.2
Dictionary File	0.1
Processing Modules	0.7

is solved in the fixed point implementation of SIFT described by Ko et al [54], who report a run time of roughly 2-2.5 seconds (shown in Table 5.2). Since the inflated SIFT processing time that we measure is an artifact of the implementation, we use the fixed point SIFT run-time numbers for the rest of this section. The table also shows the the time required for lossy and lossless compression on the iMote2, which we perform before transmitting an image.

Table 5.3 reports the memory usage of major components in our system. The arm-linux on an iMote2 takes about half of the total memory, and the image capture and SIFT processing components take about a quarter of the memory. As a result of our optimizations, the inverted index and vocabulary tree are highly compact and consume only a few hundred kilobytes of memory. The dictionary file for handling continuous queries corresponds to the images in memory.

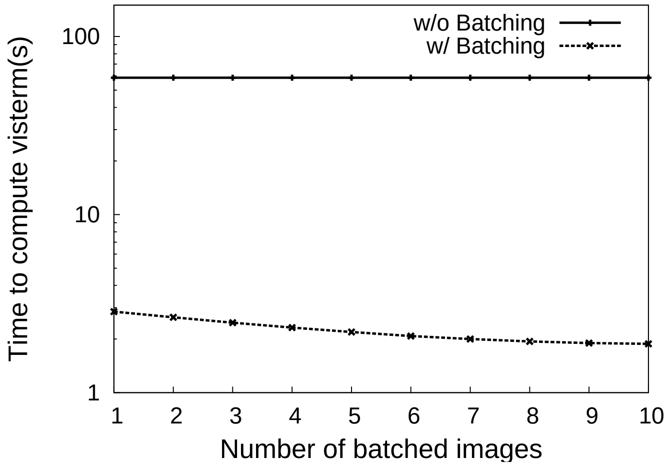


Figure 5.7. Impact of batching on lookup time.

5.9.2 Evaluation of Vocabulary Tree

In this section, we evaluate the performance of our vocabulary tree implementation and show the benefits of partitioning and batching to optimize lookup overhead.

5.9.2.1 Impact of Batched Lookup

We evaluate the impact of batched lookup on a vocabulary tree with 64K visterms, which has a depth of 4 and a branching factor of 16. The vocabulary tree is partitioned into 257 chunks including one root subtree and 256 second-level subtrees. Each of these chunks is around 20KB. We vary the batch size from a single image, which is a batch of roughly 200 SIFT features, to ten images, *i.e.* roughly 2000 SIFT features.

Figure 5.7 demonstrates the benefit of batched lookup. Note that the y-axis is in log scale and shows the time to lookup visterms for each image. The upper line shows the reference time consumed for lookup when no batching is used, *i.e.* when each SIFT feature is looked up independently, and the lower plot shows the effect of batching on per-image lookup time as the number of batched images increases. As can be seen, even batching the SIFT features of a single image reduces the lookup time by an order of magnitude. Further batching reduces the lookup time even more,

Table 5.4. Impact of size of vocabulary tree.

#Visterms	Branching Factor	Depth	Size (MB)	Lookup Time(s)	top-most Accuracy	top-3 Accuracy	top-5 Accuracy	Response Time(min.)
10000	10	4	0.76	2.01	0.73	0.84	0.87	2.15
65536	16	4	4.92	2.85	0.82	0.86	0.87	14.08
83521	17	4	6.26	3.16	0.84	0.87	0.92	17.91

and when the batch size increases from 1 to 10 images, the time to lookup an image reduces from 2.8 secs to 1.8 secs. This shows that batched lookups, and the use of partitioned vocabulary trees has considerable performance benefit. The drawback of buffering is the delay since images are processed until a batch is full. It is not hard to observe that the average delay of each image is in proportion to the batch size, if the images are periodically captured. However, for the applications that are not in strictly realtime manner, the influence of delay can be overcome by choosing a proper buffer size.

5.9.2.2 Impact of Vocabulary Size

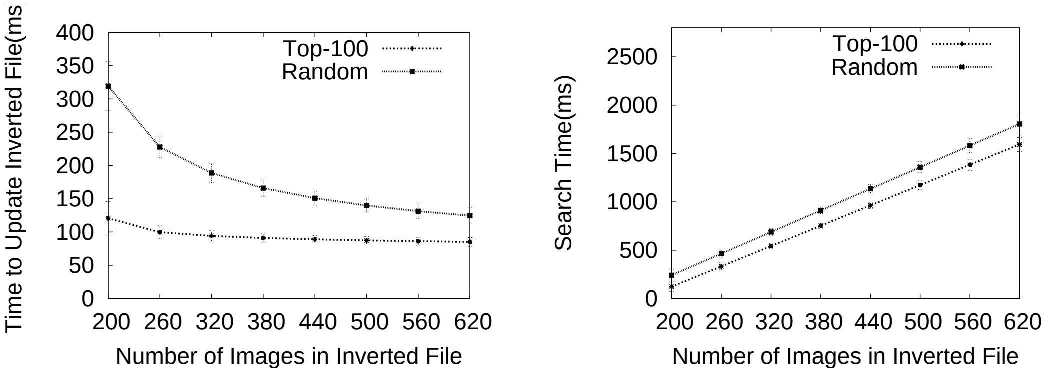
The size of the vocabulary tree has a significant impact on both the accuracy as well as the performance of search. From an accuracy perspective, the greater the number of visterms in a vocabulary tree, the better is the ability of the search engine to distinguish between different features. From a performance perspective, larger vocabularies mean greater time and energy consumed for both lookup from flash, as well as for dynamic reprogramming. To demonstrate this tradeoff between accuracy and performance, we evaluate three vocabulary trees of different sizes constructed from our training set. The accuracy and lookup numbers are averaged over twenty search queries.

Table 5.4 reports the impact of vocabulary size on three metrics: the lookup time per-image for converting SIFT feature to visterms, the search accuracy and the time to reprogram the vocabulary tree over the radio. In our evaluation, a matched result is defined as one of the top-k ranking list is correct, and accuracy is defined as the fraction of queries that have matched search results. We let k = 1,3,5 respectively in our experiment. The results show that the lookup time increases with increasing vocabulary size as expected, with over a second difference between lookup times for the vocabulary tree with 10K vs 83K nodes. The search accuracy increases with the size of the vocabulary tree as well. As the size of the vocabulary tree grows from 10K to 83K nodes, the accuracy increases by 15% from 0.73 to 0.88. In fact, the greatest gains in accuracy are made until the vocabulary tree becomes roughly 64K in size. Increasing the size of the vocabulary tree beyond 83K has a very small effect on the accuracy — a vocabulary tree with 100K nodes has an accuracy of 90% but larger trees give us no further improvement because of our small training set.

Table 5.4 also validates a key design goal — the ability to dynamically reprogram the sensor search engine with new vocabulary trees. The last column in the table shows the reprogramming time in the case of a one-hop network. A small sized vocabulary tree with 10K nodes is roughly 750 KB in size and can be reprogrammed in less than three minutes, which is not very expensive energy-wise and is feasible in most sensor network deployments. Even a larger vocabulary tree may be feasible for reprogramming, since trees with 64K and 83K nodes take about 20 minutes to reprogram.

5.9.3 Evaluation of Inverted Index

Having discussed the performance of the vocabulary tree, we turn to evaluating the core data structure used during search, the inverted index. In particular, we validate our use of the top-k list for determining which visterm sequences to save to flash. The inverted index is given approximately 1MB of memory, which is completely used once about 150 images are stored on the local sensor node. Beyond this point,



(a) Update time

(b) Search time

Figure 5.8. Inverted index performance

further insertions result in writes of the inverted index to flash. The vocabulary tree size that we use in this experiment has 64K visterms.

We compare two methods of storing the inverted index in this experiment. The first technique, labeled “Random”, is one where when the size of the inverted index increases beyond the memory threshold, a random set of 100 visterms are chosen, and their image lists are saved to flash. The second scheme, called “Top-100” is one where the indexes of the hundred longest visterm chains are maintained in a separate array in memory, and when the memory threshold is reached, the visterm lists with entries in this array are flushed.

Figure 5.8(a) reports the average time per image to update the inverted file using Random and top-100 methods. It can be seen that the update time using the top-100 method is less than half of the time for the Random scheme when there are roughly 200 images in the database and the gains reduce to about 25% when there are around 600 images. The diminishing gains with larger number of images is because the average length of the image list for each visterm increases with more images, therefore, even the Random list has a reasonable chance of writing a long list to flash.

Figure 5.8(b) presents the search time on the inverted file stored by Random and top-100 methods respectively. As the size of the inverted file grows, the search time also increases. However, the total time for search grows only up to a couple of seconds even for a reasonably large local image dataset with 600 images. The results also shows that the top-100 has less search time than random since it needs to read fewer times from flash during each search operation, although the benefits are only 10-15%.

5.9.4 Evaluation of Query Performance

We now evaluate the performance of our system for ad-hoc queries and continuous queries.

5.9.4.1 Benefits of Visterm Query

One of the optimizations in our search system is the ability to query by visterm *i.e.* instead of transmitting an entire image to a sensor, we only need to transmit visterms of the image. In this experiment, we compare the energy cost of visterm-based querying against two other variants. The first is an "Image query", where the entire image is transmitted to the sensor, which generates SIFT features and visterms from the query image locally, and performs local matching. The second scheme, labeled "SIFT query", corresponds to the case where the SIFT features for the query image are transmitted to the sensor, and the sensor generates visterms locally and performs matching. Here, we only measure the total energy cost of transmitting and interpreting the query, and do not include the cost of search to generate query results, which is common to all schemes. As shown in Table 5.5, transmitting only query visterms reduces the total cost of querying by roughly 20x and 10x in comparison with schemes that transmits the entire image or the SIFT features respectively. Much of this improvement is due to the fact that visterms are extremely small in comparison to transmitting the full image or SIFT features.

Table 5.5. Energy cost of querying (J)

Query Type	Communication	Computation	Total
Image query	3.5	0.52	4.02
SIFT query	2.45	0.04	2.49
Visterm query	0.01	0	0.01

5.9.4.2 Ad-hoc vs Continuous Query

Ad-hoc queries and continuous queries are handled differently as discussed in Section 5.7. Both queries require image capture, SIFT feature extraction, and visterm computation. After this step, the two schemes diverge. Ad-hoc query processing requires that an inverted file is updated when an image is captured, and a search is performed over the database images when a query is received. A continuous query is an image-to-image match where the captured image is matched against the images of interest for the active continuous queries.

Table 5.6 provides a breakdown of the energy cost of these components. The batch size used for the vocabulary tree is 10 images. As can be seen, our optimizations result in tremendously reduced visterm computation and search costs. Both continuous and ad-hoc queries consume less than 0.25 Joules per image. In fact, the cost is dominated by SIFT features computation (we address this in Section 5.9.6). Thus, both types of queries can be cheaply handled in our system.

5.9.5 Distributed Search Performance

Having evaluated several individual components of our search engine, we turn to an end-to-end performance evaluation of search in this section. In each experiment, the search system runs on the iMote2 testbed for an hour, and the image capture rate is set to 30 seconds. The query rate is varied for different experiments as specified. The results show the aggregate numbers for different operations over this duration.

Table 5.6. Energy cost of capturing and searching an image (J)

Component	Task	Energy (J)
Image Capture	Capture Image	0.04
Image Representation	Compute SIFT Feature	0.48
	Compute Visterm	0.04
Ad-hoc Querying	Update Inverted File	0.02
	Search	0.23
Continuous Querying	Match Visterm Histogram	0.16

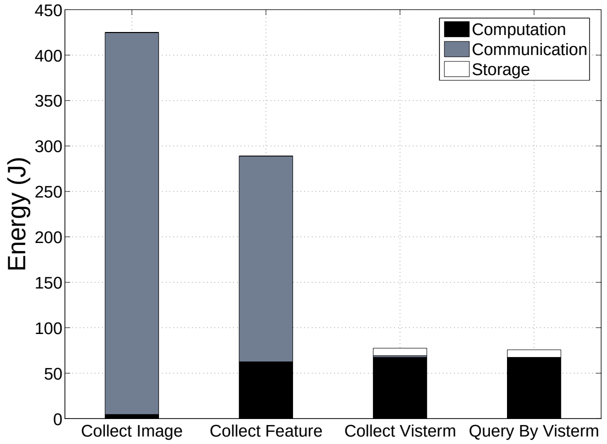


Figure 5.9. Total Energy cost of four mechanisms

5.9.5.1 Push vs Pull

We compare two approaches to design a distributed search engine for sensor networks — a push-based approach vs a pull-based approach. There are three types of push-based approaches: (a) all captured images are transmitted to the proxy, in which case there is no need for any computation or storage at the local sensor, (b) when SIFT features are transmitted, and only the SIFT processing is performed at the sensor, and (c) when visterms are transmitted, therefore both SIFT processing and vocabulary tree lookup are performed locally. In a pull-based approach, the vis-

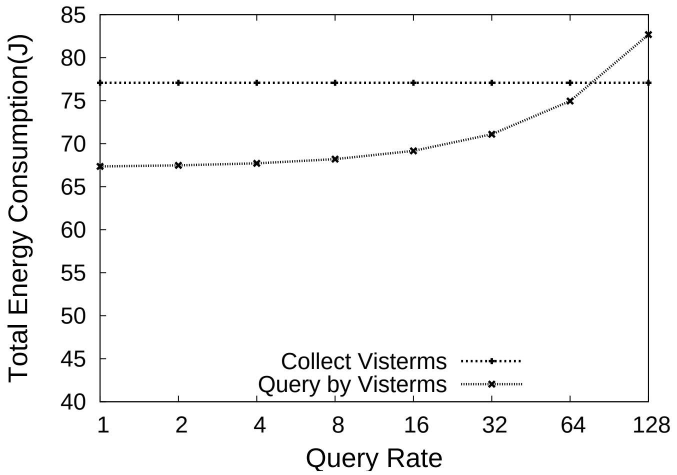


Figure 5.10. A comparison of collect visterms and query by visterms

terms corresponding to the query are transmitted to the sensors, and the query is locally processed at each sensor in the manner described in Section 5.7.

Figure 5.9 provides a breakdown of the communication, computation, and storage overhead for the four schemes — “Collect image”, “Collect features” and “Collect Visterms” are three push-based schemes, and “Query by Visterms” is the in-network search technique that we use. The query rate is fixed to four queries an hour and the sampling rate is one image per 30 seconds. As can be seen, the computation overhead is highest for the query-by-visterms scheme, but the communication overhead is tremendously reduced; storage consumes only a small fraction of the overall cost. A query-by-visterms scheme consumes only a fifth of the energy consumed by an approach that sends all images to the proxy, and only a third of a scheme that sends SIFT features.

Figure 5.9 also shows that the “Collect Visterms” and “Query by Visterms” schemes have roughly equivalent performance. We now provide a more fine-grained comparison between the two schemes in Figure 5.10. A “Collect Visterms” scheme consumes more communication overhead for transmitting visterms of each captured image but does not incur the computation overhead to maintain, update, or lookup the inverted index for a query. The results show that unless the query rate is extremely high (more than one query/min), the query-by-visterms approach is better than a collect visterms approach. However, we also see that the difference between the two schemes is only roughly 15% since visterms are very compact and not very expensive to communicate. Since both schemes are extremely efficient, the choice between transmitting visterms to the proxy and searching at a proxy vs transmitting query visterms to the sensor and searching at the sensor depends on the needs of the application. Our system provides the capability to design a sensor network search engine using either of these methods.

Notice that our evaluation is carried out on the iMote2 platform where the CPU consumes almost the same power as the radio(see Table 5.1). If we apply our paradigm to a computation cheap platform, we expect that the benefit of querying by visterms would give us even higher benefit. Meanwhile, querying by visterms is a more flexible paradigm than collecting visterms since it doesn't need extra infrastructure to provide query processing and data storage functionality. For instance, sensor nodes can directly search with each other using a "querying by visterms" paradigm without the need for a central proxy.

5.9.5.2 Multi-hop Latency

So far, our evaluation has only considered a single hop sensor network. We now evaluate the latency incurred for search in a multi-hop sensor network. We place five iMote2 nodes along a linear multi-hop chain in this experiment, and configure the topology by using a static forwarding table at each node. The total round trip latency for a user search includes: (a) the time taken by the proxy to process the query image and generate visterms, (b) latency to transmit query visterms to the destination mote via a multihop network, (c) local search on the mote, (d) transmission of the local ranked list of query results, (e) merging the individual ranked lists at the proxy,

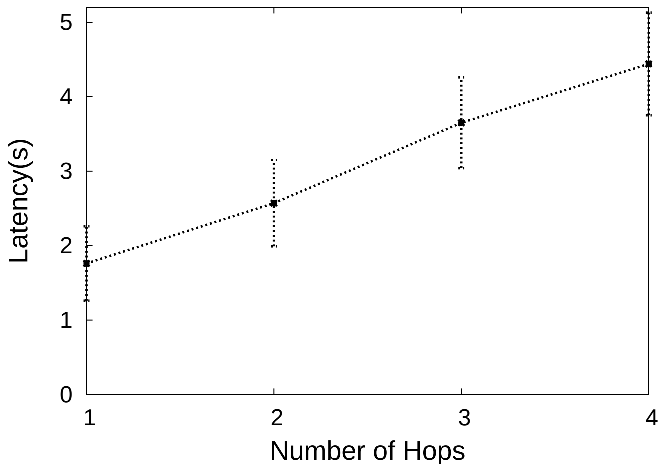
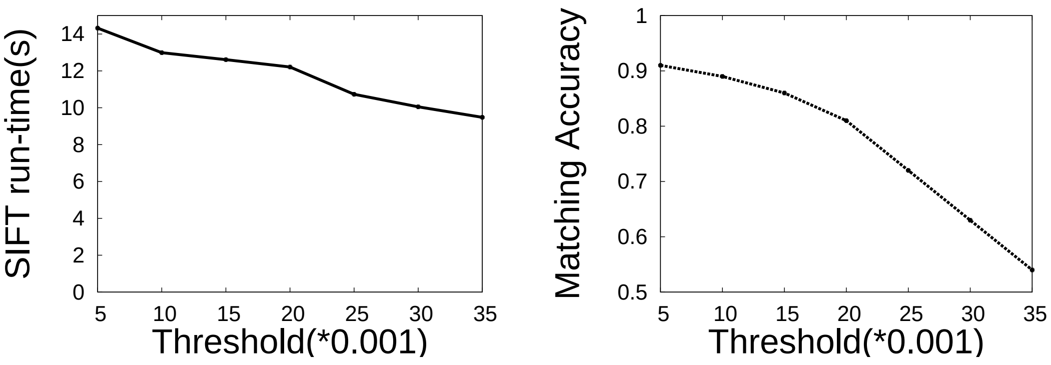


Figure 5.11. Round trip latency in multihop environment

and finally (f) transmission of the global ranked list to the sensors so that they can transmit thumbnails or full images. We do not include the time taken to transmit the thumbnails or images in this experiment, and only take into account the additional overhead resulting from performing our iterative search procedure involving multiple rounds between the proxy and the sensor. The maximum size of a ranked list that can be transmitted by any sensor is set to five.

Figure 5.11 shows the round trip latency over multiple hops for a single query. As expected, the round trip latency increases as the number of hop grows, however, even for a four hop network, the overall search latency is only around 4 seconds, which we believe is acceptable given the increased energy efficiency.

In our evaluation, we broadcast queries to all the six nodes and send back visterms to proxy. In a large scale network, queries usually have other constraints, for instance, they may be restricted to specific geometric regions, or to a specific time frame. By adopting query processing algorithms, we can direct queries to a subset of nodes thus reduce the overhead. Also we can adopt in-network aggregation algorithm to reduce the energy cost of sending back visterms from sensors to proxy. We leave this aspect for future work.



(a) Run-time(s)

(b) Accuracy

Figure 5.12. Impact of tuning SIFT threshold

5.9.6 Tuning SIFT Performance

As shown in Table 5.6, our optimizations of visterm extraction and search dramatically reduce the energy cost of these components, leaving SIFT processing as the primary energy consumer in our system. A key parameter in SIFT is the threshold that controls the sensitivity of SIFT features and we explore how to optimize it. Larger thresholds lead to fewer extracted features and vice-versa. In practice, matching accuracy increases with the number of features. The default threshold value in SIFT is 0.007, which corresponding to approximately 200 features for a QVGA image of a typical book cover.

Figure 5.12(a) shows the SIFT run-time (we use the floating point version since we do not have the optimized version) for different thresholds, and Figure 5.12(b) shows the corresponding image matching accuracy. The graphs show that as the threshold increases to 0.35, the running time of the algorithm drops by a third but the matching accuracy drops by around 30% as well. A reasonable operating region for the algorithm is to use a threshold between 0.005 and 0.02, which reduces SIFT processing time by about 2 seconds, while the accuracy is above 80%. Since these benefits are solely due to the reduction in the number of features, we believe that similar gains can be obtained with the fixed point version of SIFT.

5.10 Summary

In this chapter we have presented the design and implementation of SenSearch, a distributed search engine for smartphone camera sensor networks, and showed that such a design is both energy-efficient and accurate.

Our key contributions were five-fold. First, we designed a distributed image search system which represents images using a compact and efficient vocabulary of visterms. Second, we designed a buffered vocabulary tree index structure for flash memory that uses batched lookups together with a segmented tree to minimize lookup time. Third, we designed a log-based inverted index that optimizes for insertion by storing data in a log on flash, and optimizes lookup by writing longest sequences in flash. Fourth, we designed a distributed merging scheme that can merge scores across multiple sensor nodes. Finally, we showed using a full implementation of our system on a network of iMote2 camera sensor nodes that our system is up to five times more efficient than alternate designs for camera sensor networks.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

6.1 Conclusion

This dissertation addresses the key challenges of exploiting the sensing capability of mobile phones to enable novel mobile systems and services.

We identify two fundamental problems that prevent mobile phones from utilizing the full potential of the sensing capability. First, we observe that the personal context information sensed by mobile phones has not been used to improve the system performance of mobile phones. Second, we observe that multimedia data sensed by mobile phones, especially images, suffers from poor accuracy and energy efficiency with local processing.

To address the first problem, this thesis presents a novel application preloading engine for mobile operating systems named FALCON. FALCON exploits mobile context, especially spatial and temporal context, to infer application usage patterns from mobile users, and prelaunch applications based on those patterns. By accurately prelaunching applications ahead of the actual usage time, the responsiveness of mobile applications can be significantly improved. To address the second problem, we provide solutions with emphases on accuracy and energy efficiency respectively. We first propose an image search engine called Crowdsearch, which exploits human computation to validate and thus improve the accuracy of processing the images captured by mobile phones, while still preserve real-time responses to mobile users despite human validation is involved in the system. We then propose a distributed image search system called SenSearch, where compact features named visterms are extracted from mobile phone images, and used for search through other phones or cloud-based servers. Since visterms are two magnitudes smaller than raw images, SenSearch reduces the energy and bandwidth cost significantly.

6.2 Future Research

While the dissertation presents novel solutions for context-aware mobile operating systems and accurate and energy efficient mobile sensing systems, it also opens up abundant research opportunities. At the end of this thesis, we envision a few feasible directions of the system research of mobile computing with an emphasis on the sensing capability.

Context-aware Mobile OS: FALCON opens up several avenues in context-aware mobile operating system research. First, context information can be used for a broader range of OS services besides application responsiveness. For instance, energy management can be beneficial from user context. One feasibility is that recharge opportunities can be inferred from user context, such as the location of "at home" implies ample recharge opportunities thus the energy policy can be more aggressive by allowing more OS services running.

Crowdsourcing-based mobile services: CrowdSearch discusses how to involve realtime crowdsourcing support into mobile services such as image search. There are several potential opportunities that could improved the performance of CrowdSearch and inspire now applications. One example is to design human validation tasks to further reduce human bias and response delay from crowdsourcing services. Another feasible improvement is to use the human validation results as a feedback to improve the performance of automated image search.

Crowdsourced smartphone sensing: Our work on distributed image search for smartphones opens up a number of new opportunities for distributed sensing research. Our energy- and bandwidth-efficient search paradigm can be potentially extended to multiple sensing modalities, such as acoustic, vibration, weather or location. We also seek to explore new image representations that incorporates other features or metadata to enable novel applications.

BIBLIOGRAPHY

[1] acrossair. acrossair. `http://www.acrossair.com/apps_acrossairbrowser.htm`.

[2] Aggarwal, Charu C., Wolf, Joel L., and Yu, Philip S. Caching on the world wide web. *IEEE Trans. Knowl. Data Eng.* (1999).

[3] Aghajan, H., Augusto, J., Wu, C., McCullagh, P., and Walkden, J. Distributed vision-based accident management for assisted living. In *ICOST* (2007), pp. 196–205.

[4] Agrawal, Sandip, Constandache, Ionut, Gaonkar, Shravan, Roy Choudhury, Romit, Caves, Kevin, and DeRuyter, Frank. Using mobile phones to write in air. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 15–28.

[5] Apple. Apple - batteries - iphone. http://www.apple.com/batteries/iphone.html.

[6] Apple. Apples new maps for ios 6 features navigation, crowd-sourced traffic. `http://promacs.com/blog/?p=96`.

[7] Apple. Daily tip: How to make your iphone camera launch instantly [jailbreak]. http://www.tipb.com/2011/04/20/daily-tip-iphone-camera-launch-instantly-jailbreak/.

[8] Apple. ios 5 slowing iphone 4 and 4s complaints. http://www.phonesreview.co.uk/2011/10/25/ios-5-slowing-iphone-4-and-4s-complaints/.

[9] around, Twitter. Augmented reality twitter is the coolest thing ever. `http://mashable.com/2009/08/04/augmented-reality-twitter/`.

[10] Asano, Futoshi, Yamamoto, Kiyoshi, Ogata, Jun, Yamada, Miichi, and Nakamura, Masami. Detection and separation of speech events in meeting recordings using a microphone array. *EURASIP J. Audio Speech Music Process. 2007*, 2 (Apr. 2007), 1–1.

[11] Azizyan, Martin, Constandache, Ionut, and Choudhury, Romit. Surroundsense: mobile phone localization via ambience fingerprinting. *In Proceedings of MobiCom 09* (Sep 2009).

[12] Baeza-Yates, R., and Ribeiro-Neto, B. *Modern Information Retrieval.* ACM Press, 1999.

[13] Bao, Ling, and Intille, Stephen S. Activity recognition from user-annotated acceleration data. Springer, pp. 1–17.

[14] Bargh, Mortaza S., and de Groote, Robert. Indoor localization based on response rate of bluetooth inquiries. In *Proceedings of the first ACM international workshop on Mobile entity localization and tracking in GPS-less environments* (New York, NY, USA, 2008), MELT '08, ACM, pp. 49–54.

[15] Bellotti, Victoria, Begole, Bo, Chi, Ed H., Ducheneaut, Nicolas, Fang, Ji, Isaacs, Ellen, King, Tracy, Newman, Mark W., Partridge, Kurt, Price, Bob, Rasmussen, Paul, Roberts, Michael, Schiano, Diane J., and Walendowski, Alan. Activity-based serendipitous recommendations with the magitti mobile leisure guide. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2008), CHI '08, ACM, pp. 1157–1166.

[16] Burnham, K. P., and R., Anderson D. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach, Second Edition.* Springer Science, New York, 2002.

[17] Campbell, Andrew T., Eisenman, Shane B., Lane, Nicholas D., Miluzzo, Emiliano, and Peterson, Ronald A. People-centric urban sensing. In *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet* (New York, NY, USA, 2006), ACM, p. 18.

[18] Chart, Star. Star chart. `https://play.google.com/store/apps/details?id=com.escapistgames.starchart&hl=en`.

[19] Chen, Guanling, and Kotz, David. A survey of context-aware mobile computing research. Tech. rep., Hanover, NH, USA, 2000.

[20] Chen, Yin, Lymberopoulos, Dimitrios, Liu, Jie, and Priyantha, Bodhi. Fm-based indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 169–182.

[21] Choudhury, Tanzeem, Borriello, Gaetano, Consolvo, Sunny, Haehnel, Dirk, Harrison, Beverly, Hemingway, Bruce, Hightower, Jeffrey, Klasnja, Predrag, Koscher, Karl, LaMarca, Anthony, Landay, James A., LeGrand, Louis, Lester, Jonathan, Rahimi, Ali, Rea, Adam, and Wyatt, Danny. The mobile sensing platform: An embedded activity recognition system. *IEEE Pervasive Computing 7*, 2 (2008), 32–41.

[22] Chu, David, Kansal, Aman, Liu, Jie, and Zhao, Feng. Mobile apps: it's time to move up to condos. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (2011), HotOS'13.

[23] Chum, Ondvrej, Philbin, James, Isard, Michael, and Zisserman, Andrew. Scalable near identical image and shot detection. In *Proceedings of CIVR '07* (New York, NY, USA, 2007), pp. 549–556.

[24] CNN. Cnn report: New jersey familys picture catches theft in the making. `http://www.cnn.com/2010/CRIME/08/24/new.jersey.theft.photo/index.html?hpt=C1`.

[25] Consolvo, Sunny, McDonald, David W., Toscos, Tammy, Chen, Mike Y., Froehlich, Jon, Harrison, Beverly, Klasnja, Predrag, LaMarca, Anthony, LeGrand, Louis, Libby, Ryan, Smith, Ian, and Landay, James A. Activity sensing in the wild: a field trial of ubifit garden. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems* (2008), CHI '08, ACM, pp. 1797–1806.

[26] Cuervo, Eduardo, Balasubramanian, Aruna, Cho, Dae-ki, Wolman, Alec, Saroiu, Stefan, Chandra, Ranveer, and Bahl, Paramvir. Maui: Making smartphones last longer with code offload. In *MobiSys 2010* (2010).

[27] Dai, Hui, Neufeld, Michael, and Han, Richard. ELF: an efficient log-structured flash file system for micro sensor nodes. In *ACM SenSys '04* (2004), pp. 176–187.

[28] Directory, Yellow Page. Yellow page directory. `http://www.yellowpages.com/`.

[29] Dornbush, Sandor, Joshi, Anupam, Segall, Zary, and Oates, Tim. A human activity aware learning mobile music player. In *Proceedings of the 2007 conference on Advances in Ambient Intelligence* (Amsterdam, The Netherlands, The Netherlands, 2007), IOS Press, pp. 107–122.

[30] Eisenman, S. B., Miluzzo, E., Lane, N. D., Peterson, R. A., Ahn, G-S., and Campbell, A. T. The bikenet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th international conference on Embedded networked sensor systems* (New York, NY, USA, 2007), SenSys '07, ACM, pp. 87–101.

[31] Eisenman, Shane B., Lane, Nicholas D., Miluzzo, Emiliano, Peterson, Ronald A., seop Ahn, Gahng, and Campbell, Andrew T. Metrosense project: People-centric sensing at scale. In *In WSW 2006 at Sensys* (2006).

[32] facebook. facebook. `http://facebook.com/`.

[33] Feng, S.L., Manmatha, R., and Lavrenko, V. Multiple bernoulli relevance models for image and video annotation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2004), pp. 1002–1009.

[34] flickr. flickr. `http://www.flickr.com/`.

[35] Fogarty, James, Hudson, Scott E., Atkeson, Christopher G., Avrahami, Daniel, Forlizzi, Jodi, Kiesler, Sara, Lee, Johnny C., and Yang, Jie. Predicting human interruptibility with sensors. ACM Press, pp. 257–264.

[36] FourSquare. Foursquare. `http://foursquare.com/`.

[37] Ganti, Raghu K., Jayachandran, Praveen, Abdelzaher, Tarek F., and Stankovic, John A. Satire: a software architecture for smart attire. In *Proceedings of the 4th international conference on Mobile systems, applications and services* (New York, NY, USA, 2006), MobiSys '06, ACM, pp. 110–123.

[38] Ganti, Raghu K., Pham, Nam, Ahmadi, Hossein, Nangia, Saurabh, and Abdelzaher, Tarek F. Greengps: a participatory sensing fuel-efficient maps application. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 151–164.

[39] Gaonkar, Shravan, Li, Jack, Choudhury, Romit Roy, Cox, Landon, and Schmidt, Al. Micro-blog: Sharing and querying content through mobile phones and social participation. In *ACM MobiSys* (2008), pp. 174–186.

[40] Garlan, David, Siewiorek, Daniel P., and Steenkiste, Peter. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing 1* (2002), 22–31.

[41] Google. Google goggles. `http://googlegoggles.com`.

[42] Goshorn, R., Goshorn, J., Goshorn, D., and Aghajan, H. Architecture for cluster-based automated surveillance network for detecting and tracking multiple persons. In *ICDSC* (2007).

[43] Harrison, Beverly L., Consolvo, Sunny, and Choudhury, Tanzeem. Using multimodal sensing for human activity modeling in the real world. In *Handbook of Ambient Intelligence and Smart Environments*. 2010, pp. 463–478.

[44] Hengstler, S., Prashanth, D., Fong, S., and Aghajan, H. Mesheye: A hybrid-resolution smart camera mote for applications in distributed intelligent surveillance. In *Information Processing in Sensor Networks (IPSN-SPOTS).* (2007).

[45] Hoh, Baik, Gruteser, Marco, Herring, Ryan, Ban, Jeff, Work, Daniel, Herrera, Juan-Carlos, Bayen, Alexandre M., Annavaram, Murali, and Jacobson, Quinn. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2008), MobiSys '08, ACM, pp. 15–28.

[46] Hui, J. W., and Culler, D. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM SenSys* (2004), pp. 81–94.

[47] `http://www.intel.com/research/exploratory/motes.htm`. Intel imote2.

[48] Invasive. What's invasive? http://whatsinvasive.com/index.php/.

[49] iphoto, Apple. iphoto. http://www.apple.com/ilife/iphoto/.

[50] Joo, Yongsoo, Ryu, Junhee, Park, Sangsoo, and Shin, Kang G. Fast: Quick application launch on solid-state drives. In *FAST* (2011), pp. 259–272.

[51] junaio. junaio. http://www.junaio.com/.

[52] Kansal, Aman, Goraczko, Michel, and Zhao, Feng. Building a sensor network of mobile phones. In *IPSN* (2007), pp. 547–548.

[53] Kittur, A, Chi, E, and Suh, B. Crowdsourcing user studies with mechanical turk. *CHI 2008* (Jan 2008). Crowdsourcing applied to user study.

[54] Ko, T., Charbiwala, Z. M., Ahmadian, S., Rahimi, M., Srivastava, M. B., Soatto, S., and Estrin, D. Exploring tradeoffs in accuracy, energy and latency of scale invariant feature transform in wireless camera networks. In *ICDSC* (2007).

[55] Lane, Nicholas D., Xu, Ye, Lu, Hong, Hu, Shaohan, Choudhury, Tanzeem, Campbell, Andrew T., and Zhao, Feng. Enabling large-scale human activity inference on smartphones using community similarity networks (csn). In *Proceedings of the 13th international conference on Ubiquitous computing* (New York, NY, USA, 2011), UbiComp '11, ACM, pp. 355–364.

[56] Lester, Jonathan, Choudhury, Tanzeem, and Borriello, Gaetano. A practical approach to recognizing physical activities. In *Proceedings of the 4th international conference on Pervasive Computing* (Berlin, Heidelberg, 2006), PERVASIVE'06, Springer-Verlag, pp. 1–16.

[57] http://people.csail.mit.edu/kkl/libpmk/. LIBPMK: A Pyramid Match Toolkit.

[58] Life, Second. Your world. your imagination. http://secondlife.com/.

[59] Lifton, Joshua, and Paradiso, Joseph A. Dual reality: Merging the real and virtual.

[60] Logan, Beth, Healey, Jennifer, Philipose, Matthai, Munguia, Emmanuel, and Intille, Stephen. A long-term evaluation of sensing modalities for activity recognition. In *Proc. of Ubicomp* (2007).

[61] Lowe, D. G. Distinctive image features from scale-invariant keypoints, 2003.

[62] Lowe, D. G. Distinctive image features from scale-invariant keypoints. In *in* International Journal of Computer Vision, 60, *2004* (2004), pp. 91–110.

[63] Lu, Hong, Brush, A. J. Bernheim, Priyantha, Bodhi, Karlson, Amy K., and Liu, Jie. Speakersense: Energy efficient unobtrusive speaker identification on mobile phones. In *Pervasive'11* (2011), pp. 188–205.

[64] Lu, Hong, Pan, Wei, Lane, Nicholas D., Choudhury, Tanzeem, and Campbell, Andrew T. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys* (2009), pp. 165–178.

[65] Lu, Hong, Yang, Jun, Liu, Zhigang, Lane, Nicholas D., Choudhury, Tanzeem, and Campbell, Andrew T. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2010), SenSys '10, ACM, pp. 71–84.

[66] Lymberopoulos, Dimitrios, Bamis, Athanasios, and Savvides, Andreas. Extracting spatiotemporal human activity patterns in assisted living using a home sensor network. In *Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments* (New York, NY, USA, 2008), PETRA '08, ACM, pp. 29:1–29:8.

[67] Lymberopoulos, Dimitrios, Ogale, Abhijit S., Savvides, Andreas, and Aloimonos, Yiannis. A sensory grammar for inferring behaviors in sensor networks. In *IPSN* (2006), pp. 251–259.

[68] Lymberopoulos, Dimitrios, Riva, Oriana, Strauss, Karin, Mittal, Akshay, and Ntoulas, Alexandros. Instant web browsing for mobile devices.

[69] M. Rahimi and R. Baer and O. I. Iroezi and J. C. Garcia and J. Warrior and D. Estrin and M. Srivastava. Cyclops: In situ Image Sensing and Interpretation in Wireless Sensor Networks. In *ACM Sensys* (2005), pp. 192–204.

[70] Mathur, G., Desnoyers, P., Ganesan, D., and Shenoy, P. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *SenSys* (2006), pp. 195–208.

[71] Mathur, G., Desnoyers, P., Ganesan, D., and Shenoy, P. Ultra-low power data storage for sensor networks. In *IPSN-SPOTS* (2006), pp. 374–381.

[72] Mathur, Suhas, Jin, Tong, Kasturirangan, Nikhil, Chandrasekaran, Janani, Xue, Wenzhi, Gruteser, Marco, and Trappe, Wade. Parknet: drive-by sensing of road-side parking statistics. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 123–136.

[73] Mikolajczyk, K., and Schmid, C. A performance evaluation of local descriptors. *IEEE PAMI 27*, 10 (October 2005), 1615–1630.

[74] Miller, Harvey J., and Han, Jiawei. *Geographic Data Mining and Knowledge Discovery.* Taylor & Francis, Inc., 2009.

[75] Miluzzo, Emiliano, Cornelius, Cory T., Ramaswamy, Ashwin, Choudhury, Tanzeem, Liu, Zhigang, and Campbell, Andrew T. Darwin phones: the evolution of sensing and inference on mobile phones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 5–20.

[76] Miluzzo, Emiliano, Lane, Nicholas D., Eisenman, Shane B., and Campbell, Andrew T. Cenceme - injecting sensing presence into social networking applications. In *in EuroSSC, ser. Lecture Notes in Computer Science* (2007), pp. 1–28.

[77] Miluzzo, Emiliano, Lane, Nicholas D., Fodor, Kristóf, Peterson, Ronald, Lu, Hong, Musolesi, Mirco, Eisenman, Shane B., Zheng, Xiao, and Campbell, Andrew T. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys '08* (New York, NY, USA, 2008), ACM, pp. 337–350.

[78] Miluzzo, Emiliano, Lane, Nicholas D., Fodor, Kristóf, Peterson, Ronald, Lu, Hong, Musolesi, Mirco, Eisenman, Shane B., Zheng, Xiao, and Campbell, Andrew T. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM conference on Embedded network sensor systems* (New York, NY, USA, 2008), SenSys '08, ACM, pp. 337–350.

[79] Miluzzo, Emiliano, Pap, Michela, Lane, Nicholas D., Lu, Hong, and Campbell, Andrew T. Pocket, bag, hand, etc.- automatically detecting phone context through discovery, 2010.

[80] MobiThinking. Global mobile statistics 2012. `http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/e#lotsofapps`.

[81] Mozilla. Link prefetching. `https://developer.mozilla.org/en/link_prefetching_faq`.

[82] Mun, Min, Reddy, Sasank, Shilton, Katie, Yau, Nathan, Burke, Jeff, Estrin, Deborah, Hansen, Mark, Howard, Eric, West, Ruth, and Boda, Pter. Peir: the personal environmental impact report, as a platform for participatory sensing systems research. In *in Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys) Krakow* (2009).

[83] Nath, Suman, and Kansal, Aman. Flashdb: dynamic self-tuning database for nand flash. In *IPSN* (2007), pp. 410–419.

[84] Nilsback, M.-E. *An automatic visual Flora - segmentation and classification of flowers images.* PhD thesis, University of Oxford, 2009.

[85] Nilsback, M.-E, and Zisserman, A. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing* (2008).

[86] Nister, D., and Stewenius, H. Scalable recognition with a vocabulary tree. In *Proc. of CVPR* (2006).

[87] Philbin, J., Chum, O., Isard, M., Sivic, J., and Zisserman, A. Object retrieval with large vocabularies and fast spatial matching. In *Proc. of CVPR* (2007).

[88] picasa, Google. picasa. `http://picasa.google.com/`.

[89] Priyantha, Nissanka B., Miu, Allen K.L., Balakrishnan, Hari, and Teller, Seth. The cricket compass for context-aware mobile applications. In *Proceedings of the 7th annual international conference on Mobile computing and networking* (New York, NY, USA, 2001), MobiCom '01, ACM, pp. 1–14.

[90] Qian, Feng, Wang, Zhaoguang, Gerber, Alexandre, Mao, Zhuoqing Morley, Sen, Subhabrata, and Spatscheck, Oliver. Profiling resource usage for mobile applications: a cross-layer approach. In *MobiSys* (2011), pp. 321–334.

[91] Ravi, Nishkam, D, Nikhil, Mysore, Preetham, and Littman, Michael L. Activity recognition from accelerometer data. In *In Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence(IAAI* (2005), AAAI Press, pp. 1541–1546.

[92] Reddy, Sasank, Parker, Andrew, Hyman, Josh, Burke, Jeff, Estrin, Deborah, and Hansen, Mark. Image browsing, processing, and clustering for participatory sensing: Lessons from a dietsense prototype. In *EmNets* (2007).

[93] Rohs, Michael, Kratz, Sven, Schleicher, Robert, Laboratories, Deutsche Telekom, Berlin, Tu, Sahami, Alireza, and Schmidt, Albrecht. Worldcupinion: Experiences with an android app for real- time opinion sharing during world cup soccer games.

[94] Rose, Polar. Polar rose adds facial recognition to your flickr photos. `http://mashable.com/2009/04/22/polar-rose-2/`.

[95] Satyanarayanan, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications 8* (2001), 10–17.

[96] Sheng, Victor S., Provost, Foster, and Ipeirotis, Panagiotis G. Get another label? improving data quality and data mining using multiple, noisy labelers. In *In Proceeding of KDD '08* (2008), pp. 614–622.

[97] Shepard, Clayton, Rahmati, Ahmad, Tossell, Chad, Zhong, Lin, and Kortum, Phillip. Livelab: measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.* (2011).

[98] `http://vision.ucla.edu/~vedaldi/code/siftpp/siftpp.html`. SIFT++: A lightweight C++ implementation of SIFT.

[99] Sivic, J., and Zisserman, A. Video google: A text retrieval approach to object matching in videos. In *ICCV* (2003).

[100] Smith, Alan Jay. Design of cpu cache memories. Tech. Rep. UCB/CSD-87-357, EECS Department, University of California, Berkeley, Jun 1987.

[101] Sorokin, A., and Forsyth, D. Utility data annotation with amazon mechanical turk. *Computer Vision and Pattern Recognition Workshops* (Jan 2008).

[102] spyglass. spyglass. `http://www.macobserver.com/tmo/review/spyglass_for_ios_powerful_navigational_instrument/`.

[103] Square, Four. Four square. `http://foursquare.com`.

[104] Tag, Microsoft. Infographic: Mobile statistics, stats and facts 2011. `http://www.digitalbuzzblog.com/2011-mobile-statistics-stats-facts-marketing-infographic/`.

[105] Takacs, Gabriel, Chandrasekhar, Vijay, Gelfand, Natasha, Xiong, Yingen, Chen, Wei-Chao, Bismpigiannis, Thanos, Grzeszczuk, Radek, Pulli, Kari, and Girod, Bernd. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *Multimedia Information Retrieval* (2008), Michael S. Lew, Alberto Del Bimbo, and Erwin M. Bakker, Eds., ACM, pp. 427–434.

[106] Tan, Chiu C., Sheng, Bo, Wang, Haodong, and Li, Qun. Microsearch: When search engines meet small devices. In *Pervasive* (Sydney, Austrilia, May 2008), pp. –.

[107] tasker. Tasker. http://tasker.dinglisch.net/.

[108] Tentori, Monica, and Favela, Jesus. Activity-aware computing for healthcare. *IEEE Pervasive Computing 7*, 2 (Apr. 2008), 51–57.

[109] Tingxin Yan, Baik Hoh, Deepak Ganesan. Trucentive: A game-theoretic incentive platform for trustworthy mobile crowdsourcing parking services. In *Proceedings of the 2012 IEEE Intelligent Transportation Systems Society Conference Management System* (2012), IEEE Computer Society.

[110] twitter. twitter. `http://twitter.com/`.

[111] `http://images.google.com/imagelabeler/`. Google Labeler.

[112] `http://www.abiresearch.com/research/1002762-US+Mobile+Email+and+Mobile+Web+Access+Trends`. US Mobile Email and Mobile Web Access Trends - 2008.

[113] `http://www.google.com/mobile/products/search.html#p=default`. Goggle: Google image search on mobile phones.

[114] http://www.sensorplanet.org/. Sensor Planet: a mobile device-centric large-scale Wireless Sensor Networks.

[115] http://www.taskcn.com/. Taskcn: A platform for outsourcing tasks.

[116] http://www.topcoder.com/. www.topcoder.com.

[117] http://www.vlfeat.org/~vedaldi/code/siftpp.html. SIFT++: a lightweight C++ implementation of SIFT detector and descriptor.

[118] http://www.wired.com/gadgetlab/2008/12/amazons-iphone/. Amazon Mobile: Amazon Remember.

[119] Vinyals, Oriol, and Friedland, Gerald. Towards semantic analysis of conversations: A system for the live identification of speakers in meetings. In *Proceedings of the 2008 IEEE International Conference on Semantic Computing* (Washington, DC, USA, 2008), ICSC '08, IEEE Computer Society, pp. 426–431.

[120] von Ahn, L., and Dabbish, L. Labeling images with a computer game. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2004), ACM Press, pp. 319–326.

[121] von Ahn, Luis, Maurer, Benjamin, Mcmillen, Colin, Abraham, David, and Blum, Manuel. recaptcha: Human-based character recognition via web security measures. *Science 321*, 5895 (August 2008), 1465–1468.

[122] Wagner, Daniel, Reitmayr, Gerhard, Mulloni, Alessandro, Drummond, Tom, and Schmalstieg, Dieter. Pose tracking from natural features on mobile phones. *Mixed and Augmented Reality 0* (2008), 125–134.

[123] Wang, Haodong, Tan, Chiu C., and Li, Qun. Snoogle: A search engine for physical world. In *IEEE Infocom* (Phoenix, AZ, April. 2008), pp. –.

[124] Wang, Yi, Krishnamachari, Bhaskar, Zhao, Qing, , and Annavaram, Murali. The tradeoff between energy efficiency and. user state estimation accuracy in mobile. sensing. In *MOBICASE 2009* (2009).

[125] Wang, Yi, Lin, Jialiu, Annavaram, Murali, Jacobson, Quinn A., Hong, Jason, Krishnamachari, Bhaskar, and Sadeh, Norman. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2009), MobiSys '09, ACM, pp. 179–192.

[126] Waze. Waze. http://www.waze.com/blog/tag/social-gps/.

[127] Waze. Waze on googles crowdsourcing of traffic. http://www.waze.com/blog/tag/crowdsourcing/.

[128] Weiser, Mark. Human-computer interaction. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, ch. The computer for the 21st century, pp. 933–940.

[129] Yale. Enalab imote2 camera. `http://enaweb.eng.yale.edu/drupal/`.

[130] Yan, Tingxin, Ganesan, Deepak, and Manmatha, R. Distributed image search in camera sensor networks. *In Proceedings of SenSys 2008* (Jan 2008).

[131] yelp. yelp. `http://yelp.com/`.

[132] Zeinalipour-Yazti, Demetrios, Lin, Song, Kalogeraki, Vana, Gunopulos, Dimitrios, and Najjar, Walid. MicroHash: An efficient index structure for flash-based sensor devices. In *USENIX FAST* (2005).

[133] Zhu, Changyun, Li, Kun, Lv, Qin, Shang, Li, and Dick, Robert. iscope: personalized multi-modality image search for mobile devices. *In Proceedings of Mobisys '09* (Jun 2009).

[134] Zhu, Yunyue, and Shasha, Dennis. Efficient elastic burst detection in data streams. In *KDD '03*.