SMARTPHONE-BASED SENSING SYSTEMS FOR DATA-INTENSIVE APPLICATIONS

By

Mohammad-Mahdi Moazzami

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2017

ABSTRACT

SMARTPHONE-BASED SENSING SYSTEMS FOR DATA-INTENSIVE APPLICATIONS

By

Mohammad-Mahdi Moazzami

Supported by advanced sensing capabilities, increasing computational resources and the advances in Artificial Intelligence, smartphones have become our virtual companions in our daily life. An average modern smartphone is capable of handling a wide range of tasks including navigation, advanced image processing, speech processing, cross app data processing and etc. The key facet that is common in all of these applications is the data intensive computation.

In this dissertation we have taken steps towards the realization of the vision that makes the smartphone truly a platform for data intensive computations by proposing frameworks, applications and algorithmic solutions. We followed a data-driven approach to the system design. To this end, several challenges must be addressed before smartphones can be used as a system platform for data-intensive applications. The major challenge addressed in this dissertation include high power consumption, high computation cost in advance machine learning algorithms, lack of real-time functionalities, lack of embedded programming support, heterogeneity in the apps, communication interfaces and lack of customized data processing libraries.

The contribution of this dissertation can be summarized as follows. We present the design, implementation and evaluation of the ORBIT framework, which represents the first system that combines the design requirements of a machine learning system and sensing system together at the same time. We ported for the first time off-the-shelf machine learning algorithms for real-time sensor data processing to smartphone devices. We highlighted how machine learning on smartphones comes with severe costs that need to be mitigated in order to make smartphones capable of real-time data-intensive processing.

From application perspective we present SPOT. SPOT aims to address some of the challenges discovered in mobile-based smart-home systems. These challenges prevent us from achieving the promises of smart-homes due to heterogeneity in different aspects of smart devices and the underlining systems. We face the following major heterogeneities in building smart-homes:: (i) Diverse appliance control apps (ii) Communication interface, (iii) Programming abstraction. SPOT makes the heterogeneous characteristics of smart appliances transparent, and by that it minimizes the burden of home automation application developers and the efforts of users who would otherwise have to deal with appliance-specific apps and control interfaces.

From algorithmic perspective we introduce two systems in the smartphone-based deep learning area: Deep-Crowd-Label and Deep-Partition. Deep neural models are both computationally and memory intensive, making them difficult to deploy on mobile applications with limited hardware resources. On the other hand, they are the most advanced machine learning algorithms suitable for real-time sensing applications used in the wild. Deep-Partition is an optimization-based partitioning meta-algorithm featuring a tiered architecture for smartphone and the back-end cloud. Deep-Partition provides a profile-based model partitioning allowing it to intelligently execute the Deep Learning algorithms among the tiers to minimize the smartphone power consumption by minimizing the deep models feed-forward latency. Deep-Crowd-Label is prototyped for semantically labeling user's location. It is a crowd-assisted algorithm that uses crowd-sourcing in both training and inference time. It builds deep convolutional neural models using crowd-sensed images to detect the context (label) of indoor locations. It features domain adaptation and model extension via transfer learning to efficiently build deep models for image labeling.

The work presented in this dissertation covers three major facets of data-driven and compute-intensive smartphone-based systems: platforms, applications and algorithms; and helps to spurs new areas of research and opens up new directions in mobile computing research.

Dedicated to my beloved parents and family ...

ACKNOWLEDGMENTS

First, I thank my co-authors, more than anyone else they have influenced my view of the research process, and established in me the importance of aiming to produce quality research with the potential for impact. I would like to thank my co-advisers Guoliang Xing and Matt Mutka. I value their candid and honest opinions, their calmness and clarity of advice amid difficult times, and their patience and understanding over the past several years.

I feel fortunate to have had the opportunity to work closely with Ulrich Herberg, Daisuke Mashima and Wei-Peng Chen during one year internship at Fujitsu Research Lab. in Sunnyvale, California. I thoroughly enjoyed the chance to work with not only Ulrich, Daisuke and Wei-Peng but the many other exceptional researchers and interns at FLA. As a member of the Mobile Sensing Group and ELANS Lab at Michigan State University I count myself lucky to have been surrounded by a number of outstanding individuals who, at different stages of my PhD, have been part of the lab. In particular, I must make special mention of Dennis Philips. Over the years we have shared many long hours working together in the lab, nights and days. He is not only good colleague, but a good friend. I am definitely lucky to have had support from another amazing person over these years. I would especially like to thank Abdol Esfahanian, for being there. Over the last few years I've had so much ups and downs amid the particular family situation. Abdol was the very first and most of the time the only person I could go to. I apologize to all my family and friends for the past years. I appreciate your understanding of the unreasonably long delays in my replies to phone calls and emails. I thank you all for not giving up on me and I plan on keeping in closer contact in the future. Thank you all for your love and support.

I would like to have a special thank to my wife, then my girl-friend, Samaneh, who was always with me in all difficulties and was willing to go through them with me, like a true friend. I would

like to thank her for her calmness, for her emotional support and for her uncountable kindness. She is indeed my true friend.

Finally, I can not thank enough my parents, my brothers Manoochehr and Hamidreza, my sister, Zahra, and my sister-in-law, Azadeh, for being accepting and loving irrespective of the unpredictable nature of my grad-school life, work schedule and the uncertainty it brings. Living half way across the world complicates many things for a family and they have always stood by me, sacrificed for me and showed me there are things above material life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

Smartphones are becoming more and more embedded in our daily lives, have changed the way we interact with our environment, and the way we perform our daily activities. While early smartphones were designed to primarily support a better voice communication and basic activities like surfing web, checking emails and listing to music, technological advances helped to reduce the gap between what we consider conventional smartphones and advanced computers. As this technological divide further diminished, a new paradigm is emerging fast: smartphones are beginning to replace the so called "intelligent" or "smart" aspects of many sensing and embedded applications. The increasing computational resources offered by smartphones allow us to interface them with many other legacy and new systems directly and continuously more than ever before. Another critical component that makes smartphones a central enabler to new advances across a wide spectrum of application domains is the embedded sensors in these devices. Multi-modal sensing capability in smartphones is set to become even more critical to sensing systems as they become intertwined with existing applications such as global environmental monitoring and new emerging domains such as smart appliances and smart-home systems, sensor-based augmented and virtual reality, personal and community health-care and sport systems, and intelligent transportation systems. As such, the ubiquity of smartphones together with their multi-modal sensing capabilities

have enabled ground-breaking ways to build a wide spectrum of mobile sensing applications as never been possible before. These applications are usually *human-centric* in that the smartphone utilizes on-board sensors to sense people and characteristics of their contexts. These advances are enabled not only by rich and multiple sensing capabilities, but also by a number of other factors as well, including increased battery capacity, communications and computational resources (CPU, RAM), and new large-scale application distribution channels [Miluzzo, 2011]. By building innovative smartphone applications and embedding novel data processing algorithms to mine large scale sensing data on smartphones, it is now possible to perform scientific discoveries using ensemble of smartphones in a more cost-effective way than before [Eagle and Pentland, 2006, Moazzami et al., 2015].

Different from the *sensing-centric* applications, this dissertation considers an emerging class of smartphone-based *compute-centric* applications. In contrast to the snesing-centric nature of participatory sensing in which the application complexity mainly lies on the sensing algorithms while complex processing is offloaded to the cloud, smartphones in these applications are embedded into environments to sense and interact with the physical world autonomously over long periods of time. Applications on the smartphones are more complex and equiped with advance machine learning algortihms capable of processing large volume and complex sensor data on-the-go on-device. For instance, in the Floating Sensor Network project [Amin et al., 2007], smartphone-equipped drifters are rapidly deployed to collect real-time data about the flow of water through a river. The smartphone's GPS allows the drifter to measure volume and direction of water flow based on its real-time location and transmit the data back to the server through cellular networks. Smartphones have also been employed for monitoring earthquakes [Faulkner et al., 2011], volcanoes [VolcanoSRI 2012,], and even operating miniature satellites [NASA PhoneSat 2013,]. Another important class of smartphone-based embedded systems is cloud robots [Guizzo, 2011] [Kehoe et al., 2015]. By in-

tegrating smartphones, these robots can leverage a plethora of phone sensors to realize complex sensing and navigation capabilities and offload compute-intensive cognitive tasks like image and voice recognition to the cloud.

Compared with the traditional mote-class sensing platforms, smartphones have several salient advantages that make them promising system platforms for the aforementioned applications. These features include high-speed multi-core processors that are capable of executing advanced data processing algorithms, multiple network interfaces, various integrated sensors, friendly user interfaces and advanced programming languages. Moreover, the price of smartphones has been dropping significantly in the last decade. Many Android phones with reasonable configurations (up to 800 MHz CPU and 2 GB memory) cost less than US$50 [LG Optimus Net,].

However, several challenges must be addressed before smartphones can be used as a system platform for data-intensive applications. We face the following major challenges:

(1) High power consumption: The smartphone power management schemes are designed to adapt to user activities to extend battery time. However, they are not suitable for untethered embedded sensing systems. If the smartphone samples sensors continually, its CPU cannot enter a deep sleep state to save energy. Low-power coprocessors (e.g., M7 in iPhone5s) can handle continuous sampling, but are available on a few high-end models only.

(2) Lack of real-time functionalities: Many sensing applications have stringent real-time requirements, such as constant sampling rate and precise timestamping. However, modern smartphone OSes are not designed for meeting these real-time requirements. For instance, sensor sampling can be delayed by high-priority CPU tasks such as Android system services or user interface drawing. Our measurements show that the software timer provided by Android may be blocked by Android core system services by up to 110 milliseconds. Moreover, Android programming library does not

provide the native interfaces that allow developers to express timing requirements.

(3) Lack of embedded programming support: The programming environment of smartphone is designed to facilitate the development of networked, human-centric mobile applications. However, it lacks important embedded programming support such as resource-efficient signal processing libraries and unified primitives for controlling and communicating with peripheral accessories such as external sensors.

(4) Heterogeneity in mobile apps, communication interfaces and programming abstractions: The lack of global standards in communication, control and data management for smartphone and smart devices results in highly fragmented systems consisting of proprietary solutions provided by multiple device vendors. Users are required to use different control interfaces to interact with smart appliances in their homes or are forced to use devices sold by a single vendor to get the interconnection among the apps. Cross app scenarios also suffer from the heterogeneity in the programming abstraction owing to the lack of global standard and a dominating architecture.

(5) High computation cost in advance machine learning algorithms: Machine learning algorithms are usually computationally intensive, take considerable amount of time and resources to train and sometimes the resulting model take up a lot of space on the disk, making them difficult to deploy on resource-limited embedded systems such as smartphones and smart devices. In addition, majority of advance machine learning algorithms commonly used in real applications are supervised algorithms - require *labeled* data to train - making them difficult to use in large scale sensing applications.

In this dissertation, we propose a number of platforms, models, algorithms and applications, that advances our abilities to build smartphone-based data-intensive applications by addressing these challenges collectively. We first conduct a series of systematic measurement experiments to

study the limitations of smartphones and highlight the challenges of developing such applications. We describe the design and implementation of a generic multi-tier platform called ORBIT [Moazzami et al., 2015] that provide the inference and machine learning algorithms for data intensive continuous sensing applications deployed on off-the-shelf smartphones. Moreover, we provide case-studies of multiple different such applications built upon ORBIT and discuss the challenges addressed by ORBIT as well as opportunities provided. In addition, we describe SPOT, a platform for a new emerging class of data intensive applications in smart-home context. We describe how SPOT addresses multiple kinds of heterogeneity while providing an extensible ecosystem toward truly connected smart-homes. We describe applications built on top of SPOT and discuss how they benefit from SPOT. Finally, we describe how advance machine learning algorithms, like deep learning models, can efficiently be integrated into the smartphone applications as the core computation pipeline, by introducing Deep-Partition and Deep-Crowd-Label. Deep-Partition provides a systematic approach to analyze the architecture of deep neural networks for efficient deployment. It minimizes the feed forward execution time of deep models by partitioning them between the phone and the backend cloud subject to the application requirement, available resources and the model architecture. While Deep-Partition describes efficiency in inference time for single smartphone scenarios, this chapter also offers methods to efficiently build such deep models for sensing applications, as well as a novel method to improve the inference accuracy by using the power of crowd at inference time. It proposes these methods with a real use-case, a crowd-sourced indoor location labeling application, Deep-Crowd-Label.

1.2 Thesis Outline

The proposed smartphone sensing platforms, system architectures, algorithms, and applications in this dissertation are rigorously evaluated using a combination of analysis and experimental studies in real environments. Experimental research plays a key role in the work presented. We build large scale experimental data-intensive applications over smartphone that are optionally connected to different devices and embedded boards such as the Arduino and IOIO boards, or different smart-appliances. We study the behavior of these applications and apply our findings toward the construction of larger and more scalable smartphone-based data-intensive sensing systems. By implementing sensing systems, algorithms, and applications on off-the-shelf smartphones and leveraging large scale data processing algorithms we discover and highlight the challenges presented by realistic mobile sensing system deployments and propose solutions to address them.

1.2.1 ORBIT: A Smartphone-Based Platform for Data-Intensive Sensing Applications

In chapter 2 we present the design, implementation, and evaluation of ORBIT platform and report our experience of building data-intensive application over ORBIT in real scenarios like environmental monitoring and robotic sensing applications. ORBIT is a smartphone-based platform for data-intensive embedded sensing applications and features a tiered architecture, in which a smartphone can interface to an energy-efficient peripheral board and/or a cloud service. ORBIT as a platform addresses the shortcomings of current smartphones while utilizing their strengths. ORBIT provides a profile-based task partitioning allowing it to intelligently dispatch the processing tasks among the tiers to minimize the system power consumption. ORBIT also provides a data processing library that includes two mechanisms namely adaptive delay/quality trade-off and

data partitioning via multi-threading to optimize resource usage. Moreover, ORBIT supplies an annotation-based programming API for developers that significantly simplifies the application development and provides programming flexibility. We conduct a measurement-based profiling of the latency and power consumption of different Android smartphones and various peripheral boards. Our results suggest that, time-critical tasks, such as high-rate sensor sampling and lightweight signal processing, should be executed on the peripheral board, while compute-intensive tasks should be offloaded to the smartphone or the cloud. By intelligently dispatching the processing tasks among the smartphone, the board, and the cloud, ORBIT minimizes the system energy consumption subject to upper-bounded processing delays. ORBIT also includes a signal processing library and a component-based programming environment, which support task partitioning with embedded programming primitives. Extensive microbenchmark evaluation and three case studies including seismic sensing, visual tracking using an ORBIT robot, and multi-camera 3D reconstruction, validate the generic design of ORBIT.

1.2.2 SPOT: A Smartphone-Based Platform to Tackle Heterogeneity in Smart-Home Systems

In chapter 3 we expand the scope of our study to one of the emerging area of data-intensive sensing applications, smart-homes. In this chapter we look at the recent advancements of smart-home technologies, including broad penetration of Internet-connected smart appliances such as remotely controllable LED lights, thermostats, cameras, motion sensors, and door locks. We elaborate how these technologies have changed the way we interact with the appliances and perform our daily activities and what the challenges toward a truly connected smart-homes are. In general, the significant heterogeneity in the smart appliances has led to isolated smart-home systems in which each

single appliance vendor provides proprietary solution for appliance specific connectivity and user experience. In particular, the heterogeneity exists in different aspects of smart appliances such as control apps, communication protocol, messaging schema, data structure and variable naming. To address these challenges, we present SPOT, a user-centric, smartphone-based platform for multi-vendor heterogeneous smart-home appliances. SPOT consists of several novel mechanisms including XML and JAVA appliance driver models, annotation-based API, an appliance-adaptive user interface and appliance/state control. To validate the flexibility and generality of our approach, we have built a SPOT prototype based on 7 real appliances. Our extensive microbenchmark evaluation and case studies show that SPOT tackles different types of heterogeneities in smart appliances, significantly eases the development of cross-device smart-home applications, and improves user experience while incurring low runtime overhead. We believe SPOT is the promising solution toward a truly-connected smart-homes.

1.2.3 On-device Deep-Learning

Chapter 4 presents a novel partitioning framework for deploying deep neural models in mobile applications more efficiently. This chapter describes the design, implementation and evaluation of the Deep-Partition, which represents the first system that combines task offloading with architecture of deep neural network models. We propose a novel model partitioning framework that enables us to embed deep learning models into mobile applications by decomposing them and assigning layers of the model to different tiers based on their time-criticality, compute-intensity, and heterogeneous latency/memory consumption profiles. To this end we look into the key standard layers that deep learning frameworks provide to build any deep neural architecture. We benchmark them to achieve the layer-wise latency for each architecture. In addition, we exploit the neural networks architecture further and consider the 3D output volume of each layer and the encoding and

sparsity of each output volume. These factors drive the design of Deep-Partition. To validate the performance of this framework, we build three major representative deep neural models, partition them with Deep-Partition under several conditions. We show how Deep-Partition minimizes the end-to-end execution time of embedded deep neural models. In addition, this chapter embodies an application of deep neural models in a crowd-assisted system for location labeling. Deep-Crowd-Label uses crowd-sourcing in both training and execution and shows how crowd-sourcing architecture can be leveraged to decrease the uncertainty in the prediction of sensing pipelines. It presents novel model adaptation and transfer learning mechanism to build deep neural models for mobile application more efficiently especially when proper training data is not available. We believe methods provided by Deep-Partition and Deep-Crowd-Label significantly facilitate building high performance mobile sensing applications.

1.3 Thesis Contribution

Herein, in this dissertation, we make several broad contributions to the field of smartphone-based sensing and data-intensive sensing applications, as summarized in the following.

1- The work in this dissertation contributes to spearheading the emerging area of data-intensive sensing applications and provides generic and universal platforms for such applications. In Chapter 2 we conduct systematic measurement and modeling to understand the opportunities as well as the challenges for using smartphones for data-intensive embedded sensing applications. Our measurement results are also useful for the design of a broad class of smartphone-based sensing systems. Second, we provide an implementation of several data processing algorithms as a library as well as several mechanisms that improve the efficiency of data processing algorithms for smartphones and mechanisms to extend the hardware plat-

form by extension-boards like the Arduino board, if needed. To our best knowledge, ORBIT is the first general-purpose, extensible, application-aware, and end-to-end sensing and processing platform for smartphones-based data-intensive embedded applications. Lastly, we demonstrate the generality and flexibility of ORBIT as a platform by presenting our experience in prototyping three applications upon ORBIT: seismic sensing, multi-camera 3D reconstruction and robotic sensing. The flexible task partitioning and dispatching framework allows ORBIT to adapt to different task structures, application deadlines, and communication delays.

2- In Chapter 3 we perform a systematic study to understand the characteristics of smart-home appliances as well the opportunities and challenges for using smartphone as the central gateway to control smart-home appliances. The result of our study shows multiple aspects of heterogeneity in smart appliances. Second, we provide a flexible, extensive and extensible device driver model that supports a number of smart appliances available on the market. The driver model addresses multiple types of heterogeneity observed in our study. Third, we provide the design and implementation of the proposed platform as a smart-home system that loads the drivers at runtime along with a dynamic user interface adaptive to the features of each appliance. Lastly, we demonstrate the generality and flexibility of our system by presenting our experience in prototyping the drivers for several real appliances as well as a cross-device home application. We also discuss examples of other home applications that we have prototyped on top of our platform. We believe this work is a crucial solution for the current highly fragmented smart-home systems and is a major step toward having a truly connected smart-home.

3- In Chapter 4 we present a collection of methods to address training and execution chal-

lenges of mobile sensing pipelines embedding deep neural models, one of the most compute-intensive data processing methods for mobile application. In this chapter we address both system and algorithmic challenges in two different yet complementary perspectives: a) building the the processing pipeline, b) runtime execution of the pipeline. The methods provided in this chapter enables researchers and developers to build more efficient mobile sensing applications with built-in more accurate data processing pipelines.

We believe that ORBIT, SPOT, Deep-Partition and Deep-Crowd-Label significantly advance the understanding of opportunities and challenges in the design of smartphone-based data-intensive sensing systems. By proposing some early solutions to tackle these challenges, and ways to seize the opportunities provided, this dissertation opens up new research directions in this emerging area.

Chapter 2

ORBIT: A Smartphone-Based Platform for Data-Intensive Sensing Applications

2.1 Introduction

Owing to the rich processing, multi-modal sensing, and versatile networking capabilities, smartphones are increasingly used to build data-intensive embedded sensing applications. However, various challenges must be systematically addressed before smartphones can be used as a generic embedded sensing platform, including high power consumption, lack of real-time functionality and user-friendly embedded programming support.

In this Chapter, we take the first step toward addressing these challenges collectively. We present ORBIT, a smartphone-based platform for embedded sensing systems. In particular, ORBIT leverages off-the-shelf smartphones to meet the energy-efficiency and timeliness requirements of data-intensive embedded sensing applications. ORBIT is based on a tiered architecture that comprises up to three tiers: the cloud, the smartphone, and one or more energy-efficient peripheral boards (referred to as extBoard) that are interfaced with the smartphone. A number of extBoard platforms are currently available, such as Arduino [Arduino Board,] and IOIO [IOIO for Android,]. Therefore, if the built-in sensors on the smartphones are not suitable for sensing applications, these boards can readily integrate various accessories, such as external sensors, to an Android

phone via USB or bluetooth interface. We conduct a measurement study on the latency and power consumption of Android smartphones and extBoard platforms. Our results show that the two platforms have highly heterogeneous but complementary power/latency profiles: smartphone features higher energy efficiency due to its faster processing capability while yielding poor timing accuracy due to the overhead of OS. These results have important implication for efficient task partitioning. In particular, while the smartphone and cloud should handle long-running compute-intensive tasks, time-critical functions such as high-rate sensor sampling and precise event timestamping must be shifted to the extBoard owing to its hardware timers and efficient interrupt handling.

Motivated by the above observations, we propose a task partitioning framework that assigns tasks to different tiers based on their time-criticality, compute-intensity, and heterogenous latency/power consumption profiles. Furthermore, to take advantage of the increasing availability of multiple cores on smartphones, ORBIT implements a data partitioning scheme that decomposes matrix-based computation into multiple threads. ORBIT also integrates a data processing library that supports high-level Java annotated application programming. The design of this library facilitates the resource management of the embedded applications by promoting a delay/quality trade-off mechanism. To enable dynamic task dispatch and runtime task profiling, we develop an ORBIT runtime environment consisting of task controllers running on each tier. These controllers coordinate task execution through a unified messaging protocol. Owing to these features, ORBIT is a powerful system toolkit to build a wide spectrum of data-intensive embedded sensing applications.

Contributions of our work outlined as follows. First, we conduct systematic measurement and modeling to understand the opportunities as well as the challenges for using smartphones for data-intensive embedded sensing applications. Our measurement results are also useful for the design of a broad class of smartphone-based sensing systems. Second, we provide an implementation of several data processing algorithms as a library as well as several mechanisms that improve

the efficiency of data processing algorithms for both the smartphone and the extension board. Several components of ORBIT bear some similarity with existing embedded system platforms [Cuervo et al., 2010a, Girod et al., 2004, Newton et al., 2009, Sorber et al., 2005]. However, to our best knowledge, ORBIT is the first general-purpose, extensible, application-aware, and end-to-end sensing and processing platform for smartphones-based data-intensive embedded applications.[1] Lastly, we demonstrate the generality and flexibility of ORBIT as a platform by presenting our experience in prototyping two applications upon ORBIT: seismic sensing and multi-camera 3D reconstruction. The flexible task partitioning and dispatching framework allows ORBIT to adapt to different task structures, application deadlines, and communication delays. The experiments show ORBIT reduces energy consumption by up to 50% compared to baseline approaches.

2.2 Related Work

Mobile sensing based on smartphones has recently received significant interests. Most studies focus on the issues related to human-centric context, including coordination among multiple concurrent sensing applications [Kang et al., 2008, Kang et al., 2010, Ju et al., 2012] and sensing algorithms such as context classifiers [Chu et al., 2011]. Recently, smartphones have been used in a number of embedded sensing applications. In [Faulkner et al., 2011], smartphones are used to build an earthquake early warning system using an onboard accelerometer. In the Floating Sensor Network project [Floating sensor network project,], smartphone-equipped drifters are deployed to monitor waterways and collect real-time volume and direction of water flow based on the phone's GPS. The NASA PhoneSat project [NASA PhoneSat 2013,] has launched low-cost satellites equipped with Android smartphones. Controlled by a smartphone, such small satellites

[1] The source code of ORBIT is available at https://github.com/msu-sensing/ORBIT

could perform various tasks such as earth observation and space debris tracking. Several recent efforts focus on building *cloud robots* [Guizzo, 2011] that integrate smartphones with robots. The phone's built-in sensors are used for sensing and navigation, while compute-intensive tasks like image and voice recognition are offloaded to the cloud.

Various task offloading schemes for smartphones have been developed recently. Spectra [Flinn et al., 2002] allows programmers to specify task partitioning plans given application-specific service requirements. Chroma [Balan et al., 2003] aims to reduce the burden on manually defining the detailed partitioning plans. Medusa [Ra et al., 2012] features a distributed runtime system to coordinate the execution of tasks between smartphones and cloud. Turducken [Sorber et al., 2005] adopts a hierarchical power management architecture, in which a laptop can offload lightweight tasks to tethered PDAs and sensors. While Turducken provides a tiered hardware architecture for partitioning, it relies on the application developer to design a partitioned application across the tiers to achieve energy efficiency.

Different from these task partitioning schemes, ORBIT dispatches the execution of sensing and processing tasks in a smart-phone-based multi-tier architecture to achieve *data-intensive* applications requirements. ORBIT maximizes the battery lifetime subject to the application-specific latency constraints. Moreover, in order to support fine-grained task partitioning across the tiers, the developer specifies the application's task structure as well as real-time requirements via either Java annotations or an XML-based application model provided by ORBIT. ORBIT also provides a messaging interface to support unified data passing mechanism between heterogenous tiers and between different application components. The details of this messaging protocol is described in a technical report [Moazzami et al., 2013].

The MAUI system [Cuervo et al., 2010a] enables a fine-grained offloading mechanism to prolong the smartphone's battery lifetime. However, MAUI relies on the properties of the Microsoft

.NET managed code environment to identify the functions that can be executed remotely. When a function is executed remotely, MAUI assumes the energy associated with its local execution is saved. In contrast, ORBIT does not rely on any language specific environment and its measurement-based power profiles account for many realistic power characteristics such as CPU sleep, wake up and tail time.

The Wishbone system [Newton et al., 2009] also features a task dispatch scheme. Unlike Turducken, Wishbone uses a profile-based approach to find the optimal partition. It only considers two tiers: in-network and on-server. Unlike MAUI, Wishbone relies on the timing profile only and does not account for the power consumption. ORBIT differs from Wishbone in several ways. Wishbone uses the CPU and network timing profiles only to find the optimal task partition, while ORBIT considers the measured latency and power consumption, which leads to more energy-efficient task partitions. Moreover, Wishbone depends on the timing profiles based on sample data under the assumption that the sample data can represent actual runtime data. However, our measurement study shows that the signal processing timing profiles can exhibit significantly variations in real scenarios. To address this, ORBIT measures the statistical timing profiles at runtime, and periodically refines the partitioning results. Moreover, Wishbone formulates the partitioning problem as a 0/1 integer linear programming problem and thus supports two tiers only. In contrast, ORBIT formulates the problem as a non-linear optimization problem and supports three or more tiers.

RTDroid [Yan et al., 2014] tackles the lack of hard real-time capability of Android system and addresses the problem by redesigning and replacing several Android components in Dalvik, e.g., Looper-Handler and Alarm-Manager. In contrast, ORBIT requires no changes to the Android system. ORBIT accounts for statistical properties of task execution, and finds the best execution assignment by its task partitioning mechanism. Hence, although RTDroid and ORBIT address different sets of issues, they are complementary. In fact, ORBIT can run on RTDroid and the

ORBIT-based sensing applications can benefit from both.

Similar to ORBIT, EmStar [Girod et al., 2004] provides an environment to implement distributed embedded systems for sensing applications based on Linux-class Microservers. However ORBIT takes one major step further and proposes a design based on smartphones for the purpose they are not originally designed for, which is embedded systems. This difference in underlying technology leads to totally different design and implementation. Although EmStar and ORBIT have similar modular designs, unlike ORBIT, EmStar does not have any partitioning mechanism and it is not strictly tiered. More importantly, ORBIT provides a library of data processing algorithms that are efficient on the resource-constrained smartphone and extension board. This is not a design goal of EmStar.

2.3 Motivation and System Overview

In this section, we discuss the motivation of using smartphone as a system platform for data-intensive embedded sensing applications and the design objectives of ORBIT.

2.3.1 Motivation and Challenges

Mote-class sensing platforms such as TelosB have been widely adopted by embedded sensing applications in the past decade. However, due to the limited processing and storage capabilities, they are ill-suited for high-sampling-rate sensing applications. Recently, several single-board computers such as Gumstix [Gumstix,], SheevaPlug [Marvell Sheevaplug,], and Raspberry Pi [Raspberry Pi,], which are equipped with rich processing and storage capabilities, have been increasingly used in embedded applications. However, their designs are not particularly optimized for low-power sensing. Moreover, without on-board sensors and wireless interfaces, they need to be equipped with various peripherals for different applications.

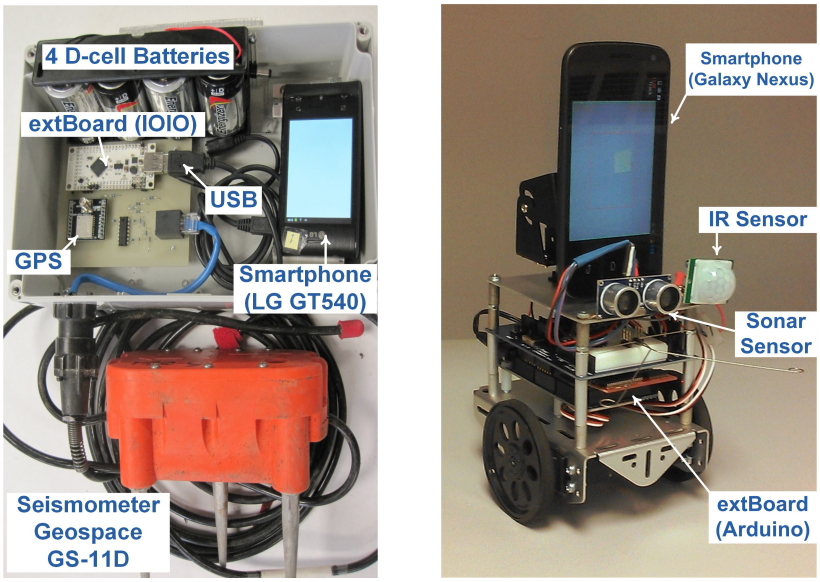
Different from the above platforms, commercial off-the-shelf smartphones offer several salient advantages that make them a promising system platform for data-intensive embedded sensing applications. The advantages include rich computation and storage resources, multiple network interfaces and sensing modalities, increasing available multi-core architecture and low cost. Moreover, smartphones come with advanced programming languages and friendly user interfaces, such as touch screen to enable rich and interactive display, unlike the limited user interfaces of motes and embedded computers (e.g., LED and buttons).

However, we still face the following major challenges in building an embedded sensing platform based on COTS smartphones:

(1) High power consumption: The smartphone power management schemes are designed to adapt to user activities to extend battery time. However, they are not suitable for untethered embedded sensing systems. If the smartphone samples sensors continually, its CPU cannot enter a deep sleep state to save energy. Low-power coprocessors (e.g., M7 in iPhone5s) can handle continuous sampling, but are available on a few high-end models only.

(2) Lack of real-time functionalities: Many sensing applications have stringent real-time requirements, such as constant sampling rate and precise timestamping. However, modern smartphone OSes are not designed for meeting these real-time requirements. For instance, sensor sampling can be delayed by high-priority CPU tasks such as Android system services or user interface drawing. Our measurements show that the software timer provided by Android may be blocked by Android core system services by up to 110 milliseconds. Moreover, Android programming library does not provide the native interfaces that allow developers to express timing requirements.

(3) Lack of embedded programming support: The programming environment of smartphone is designed to facilitate the development of networked, human-centric mobile applications. However,



(a) Seismic ORBIT node (b) Robotic ORBIT node

Figure 2.1: ORBIT nodes for seismic sensing and robots.

it lacks important embedded programming support such as resource-efficient signal processing libraries and unified primitives for controlling and communicating with peripheral accessories such as external sensors.

2.4 System Overview

In this paper, we present ORBIT, which is designed to address the above three major challenges. An ORBIT node comprises an Android smartphone, an extBoard (e.g., IOIO [IOIO for Android,] and Arduino [Arduino Board,]), and possibly a runtime system on the cloud. The extBoard is connected to the smartphone through a USB cable or bluetooth for communication. It is equipped with a low-power MCU, e.g., ATmega2560 with 16 MHz frequency, 8 KB RAM, and an analog-to-digital (A/D) convertor that can integrate various analog sensors. Fig. 2.1 shows two ORBIT

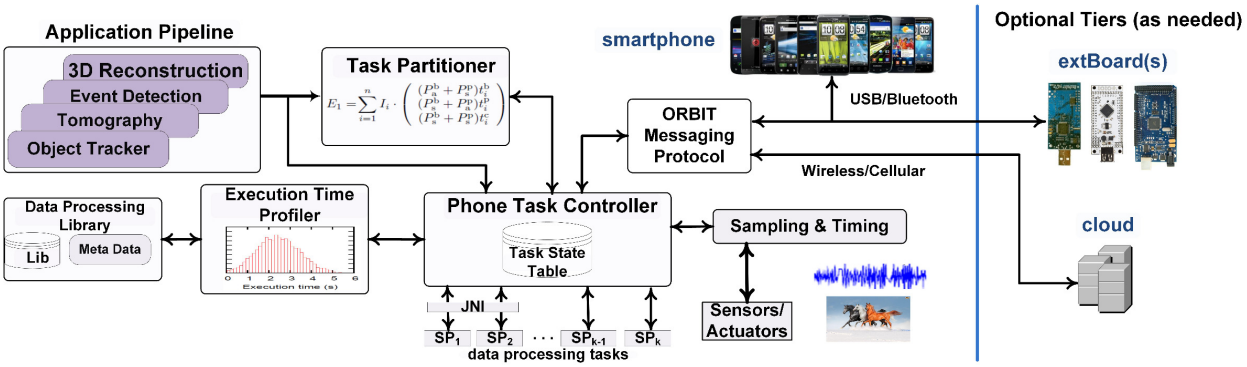


Figure 2.2: System Architecture of ORBIT.

prototypes, a seismic monitoring node and a robot sensing node that are used in the evaluation (cf. Section 3.7). Fig. 2.2 shows the overall system architecture of ORBIT.

ORBIT is designed to meet the following three requirements. (1) Energy-efficiency and while taking into account the timeliness requirements: ORBIT leverages the heterogeneous power/latency characteristics of multiple tiers (e.g., extBoard, smartphone and cloud server) to minimize the overall energy consumption. It also models the timing latency of the application statistically and applies these models in task partitioning and execution. We note that ORBIT cannot achieve hard real-time guarantees. However, the statistical task timing model allows the task deadlines to be met with higher probability. (2) Programmability: ORBIT provides a component-based programming environment that allows developers to build sensing applications without the need to deal with low-level issues of the system design. (3) Compatibility: ORBIT relies solely on the out of the box functionality of COTS smartphones, without requiring kernel-level customization or device rooting. This not only minimizes the burden on the application developers, but also ensures the compatibility with diverse smartphone models. In the following, the major ORBIT components are described.

ORBIT Library and Application Model: ORBIT provides a library of signal processing algorithms with unified interfaces. They can be easily composed into various advanced sensing applications. The library provides a programming primitive, referred to as *connection*, allowing programmers to specify application composition in an XML file or through Java annotations. In particular, each algorithm can be executed on any tier, enabling flexible task dispatching.

Task/Data Partitioner and Execution Time Profiler: To meet the deadlines of sensing applications, time-critical tasks should be executed on the extBoard while the compute-intensive tasks should be executed on the smartphone and/or the cloud. We formally formulate a task partitioning problem that aims to minimize the energy usage of the smartphone subject to a processing delay bound on time-critical tasks. Task Partitioner solves this problem and obtains the optimal task dispatch plan. A challenge presented by this design is that the signal processing tasks may have highly variable execution time. We design an online profiler that measures task execution time at runtime and runs the task partitioner dynamically. Moreover, ORBIT adopts a data partitioning scheme that decomposes matrix-based computation into multiple threads to take advantage of the increasing availability of multiple cores on smartphones.

Task Controllers and Unified Messaging Protocol: At runtime, the Task Controllers on different tiers collaboratively instantiate the tasks and execute them by following the task dispatch plan. The extBoard runs low-level and real-time functions such as sensor sampling and lightweight signal processing tasks. The smartphone and cloud run compute-intensive tasks that require data from a single and multiple ORBIT nodes, respectively. To facilitate such flexible task dispatching and control, we develop a unified messaging protocol for the communication across different tiers on top of native communication channels such as USB (between phone and extBoard) and HTTP (between phone and cloud server). Due to space constraint, the details of the messaging protocol

are omitted in this paper and can be found in a technical report [Moazzami et al., 2013].

2.5 Measurement-Based Latency and Power Profiling

To use smartphones as a system platform for data-intensive sensing applications, it is important to understand the characteristics of their latency and power consumption. This section presents a measurement study of the latency and power consumption on different smartphones. The measurement study provides insights into the limitations of smartphones and motivates several key design decisions in ORBIT. For instance, the design of the task partitioner, execution time profiler, adaptive delay/quality trade off in the library are based on the findings of the measurement study discussed in this section.

2.5.1 Timing Accuracy and Latency Profiling

Timing accuracy is critical for many sensing applications. For instance, acoustic or seismic source localization [Liu et al., 2013] typically requires millisecond level precision for the timestamps of sensor samples. In this section, we measure the accuracy of *software timer* and *event timestamping* of Android smartphones and discuss the impact on the design of ORBIT. First, an event timer is commonly used to implement constant-rate sensor sampling and its accuracy determines the sampling rate precision that can be supported. Second, timestamping an external event, which may be triggered by a GPS receiver or a sensor connected to the smartphone through USB, is also essential for many embedded applications. Our measurements are conducted using an LG GT540, a Nexus S, and a Galaxy Nexus, representing three typical low- to medium-end smartphone models. They run three versions of Android distribution, 2.1, 4.0.4, and 4.2.2. The LG GT540 results discussed here are representative of these phones measured in terms of the level of timing variability.

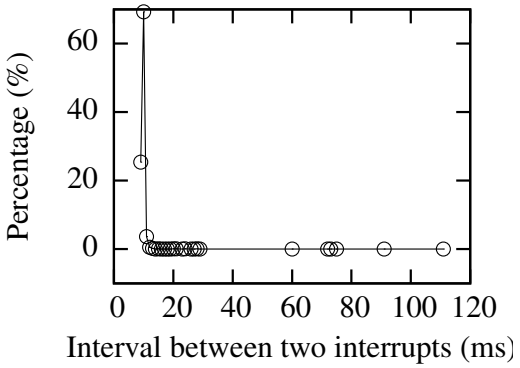


Figure 2.3: Distribution of the intervals between two interrupts raised by a software timer of Android.

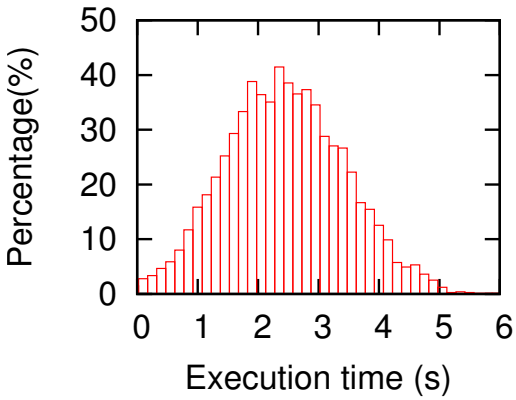


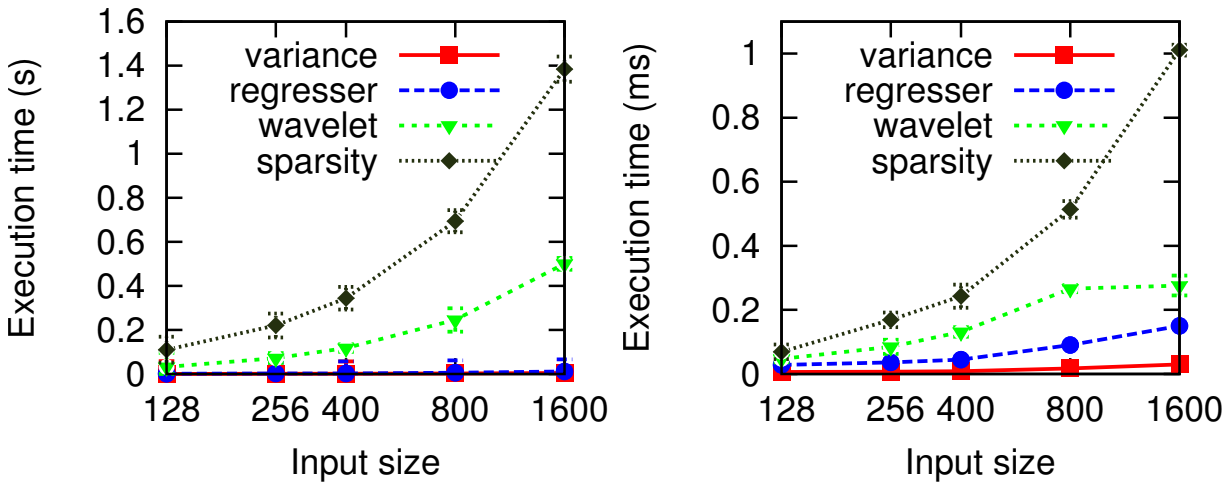
Figure 2.4: Distribution of execution time of the SIFT algorithm on 640x480 images on Nexus S.

Software Timer: Fig. 2.3 plots the distribution of the intervals between two interrupts generated by a software periodic timer with an desired interval of 10 ms, while only Android core system services are running. Although most intervals are close to 10 ms, the distribution has a long tail with a maximum interval above 110 ms.

Event timestamping: We then measure the delay between the time instance when a pulse signal is received by a digital pin of an extBoard (which triggers a USB interrupt to Android) and when the USB interrupt is received in an Android application. Our measurement shows that this delay is highly variable and can be up to 5 ms.

Due to the Android's poor timing accuracy suggested by these results, it is difficult to implement high-constant-rate sensor sampling or precise event timestamping. In contrast, our measurement shows that the timing error of an Arduino extBoard is no greater than 12 μs, due to the availability of hardware timers and efficient interrupt handling.

We then investigate the execution time of the signal processing algorithms. We find that most algorithms have relatively constant execution times for fixed input sizes. However, the execution time of a few algorithms depends on the input data. Fig. 2.4 shows the distribution of the exe-



(a) On Arduino

(b) On Nexus S

Figure 2.5: Execution time of signal processing algorithms (error bar: standard deviation).

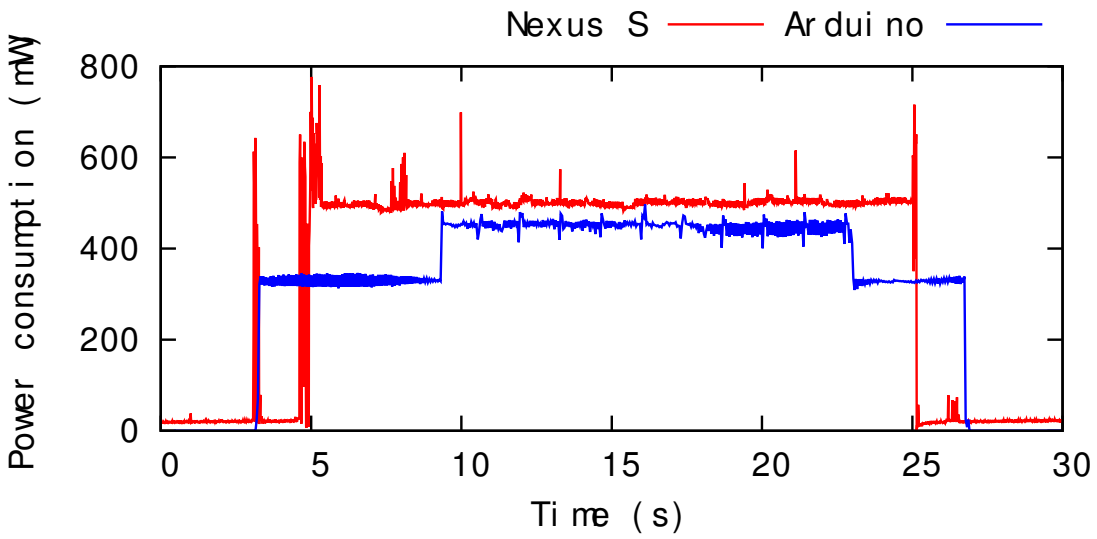
cution time of the scale-invariant feature transform (SIFT) used for detecting features of different 640x480 pixel images on a Nexus S. This example suggests that the statistical properties of the signal processing delays must be accounted for at *runtime* to ensure the real-time performance of the application.

The execution times of the tasks determine their energy consumption and highly affect the real-time performance of the application. Fig. 2.5 plots the execution times of four signal processing algorithms on an Arduino extBoard and a Nexus S smartphone versus the length of the input signal. It can be seen that extBoard's and smartphone's latencies are in the order of seconds and milliseconds, respectively. However, they have comparable power consumption as will be shown in Section 2.5.2. Therefore, the smartphone can process the signals with less energy and shorter delays.

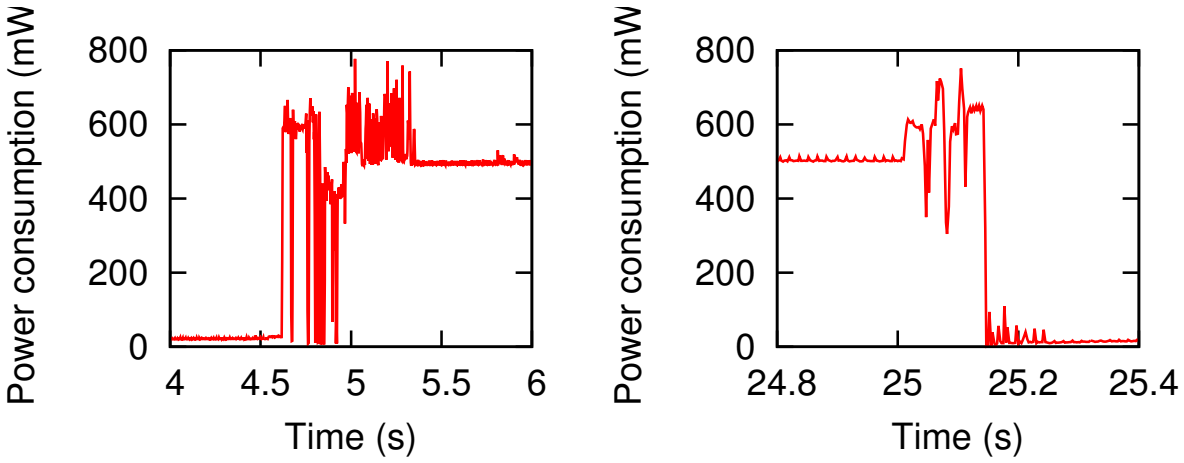
2.5.2 Power Profiling

As computation is typically the dominant source of power consumption in data-intensive sensing applications, we focus on profiling the CPU of smartphones. Power consumption of other components (e.g., radio) can be easily integrated with the measured CPU power profiles. We measure the current draw of several Android smartphones using an Agilent 34411A Multimeter. Fig. 2.6a shows the the power consumption of a Samsung Nexus S and Arduino board in different processing states. We observed similar CPU state transitions and power consumption characteristics across multiple smartphone models. Initially, the smartphone is in the sleep state, and hence draws little current (less than $5\,\text{mA}$). At the 5th second, the extBoard requests the smartphone to execute an FFT algorithm. Upon receiving the request, the phone first acquires a wake-lock, the Android mechanism to prevent the phone from going to sleep. At the 25th second, FFT completes and releases the wake-lock. Before the phone fully wakes up or goes to the sleep state, there is a transitional phase with a few power spikes. Fig. 2.6b shows the expanded view of these two transitional phases. We refer to them as *wake-up* and *tail* phases, lasting approximately 200 ms and 755 ms, respectively. There are also two spikes in Fig. 2.6a, caused by the communications between the phone and the extBoard. Since these spikes are very short and have limited current draw, their energy consumption is negligible. Based on these results, we define four CPU states: *sleep*, *wake-up*, *active* and *tail*.

The Arduino extBoard has three states, *active*, *idle*, and *sleep*. Its average current draw in these states are $90\,\text{mA}$, $66\,\text{mA}$, and near zero, respectively. In contrast to the smartphones, the transitional states for Arduino are very short (in the order of μs) and hence their energy consumption is negligible.



(a) Power consumption comparison of Arduino and Nexus S in a 30-second experiment.



(b) Zoomed-in view of the wake-up and tail states of Nexus S.

Figure 2.6: Arduino and Nexus S power consumption profiles.

2.5.3 Summary

The above profiling results show the significant heterogeneity in the power and latency profiles of different tiers (extBoard and smartphone). Although similar measurement studies have been reported in literature [Newton et al., 2009, Cuervo et al., 2010a], we collectively report our mea-

surement results and show how these findings provide important implications for both challenges and opportunities in the design of ORBIT. First, as the Android system has poor timing accuracy, time-critical functions such as high-rate sensor sampling and precise sensor event timestamping must be shifted to the extBoard owing to its hardware timers and efficient interrupt handling. Second, signal processing algorithms may have dynamic execution times, which need online profiling to ensure that the critical time deadlines of the application are met. Third, smartphones have much lower latency and higher energy efficiency than the extBoard. However, if the extBoard must stay active to continually sample sensors, it is desirable to utilize its spare time to process signals, such that the smartphone can sleep to save energy. Lastly, the transitional phases (wake-up and tail) and the data transfers among the tiers incur non-negligible overhead in both energy consumption and latencies. When dispatching signal processing tasks to different tiers, these important characteristics must be carefully considered in order to minimize the total system energy consumption while meeting application latency constraints.

2.6 Design And Implementation

This section presents the design of ORBIT to achieve the objectives discussed in Section 2.3.1.

2.6.1 Application Pipeline

An ORBIT application pipeline can be represented by a graph, where the nodes are the processing tasks and the edges are the data flows. The application pipeline, which defines the sequence of executing the tasks, is used by the component-based programming model and task partitioning module of ORBIT. Each task implements an elementary sensing or processing operation, such as computing mean, FFT or converting an image to grayscale. For example, an application pipeline can be:

sample the sensor (camera) $\rightarrow$ *low pass filter* $\rightarrow$ *face recognition* $\rightarrow$ *write into file*.

Each task itself can be made of a few smaller tasks. Such an application model offers two benefits. First, by the notion of task, we can build the latency profile of each task (as explained in section 2.5.1) and use it for task partitioning (as described in the next section). Second, ORBIT application model can significantly simplify the application development and reduce the user effort to create an application, especially for those who are not familiar with embedded system design. In particular, ORBIT presents application developers with a single programming abstraction without burdening them with low-level details such as where and how the tasks are executed and how they communicate across different tiers.

ORBIT supports two methods for specifying an application. An application developer can either write Java code using the ORBIT API or write an XML file. In either way the application pipeline specifies what tasks are used, what parameters for each task are set, and how the task are connected to form the pipeline. From this point forward, we will use a running example, shown

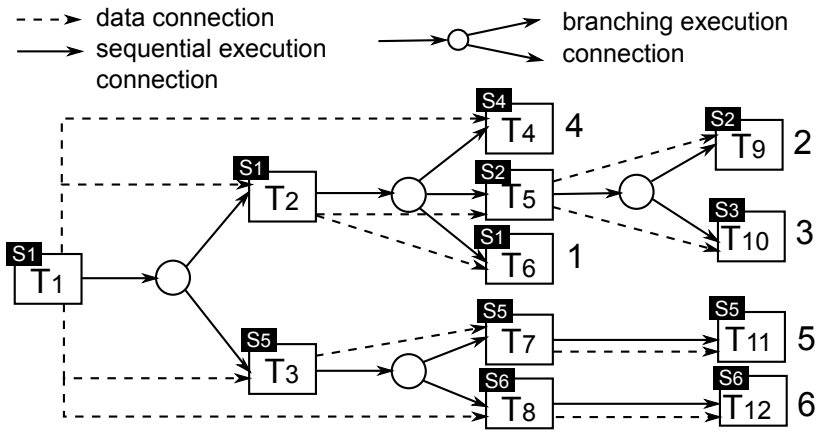


Figure 2.7: An example ORBIT application. The numbers besides the leaf nodes in the execution tree are the priorities assigned by the application developer; the tag S_x of a task represents the set it belongs to in the task partitioning solution.)

in Fig. 2.7, to illustrate how tasks are connected to build an application, as well as the automatic execution optimization and manipulation in later sections. The sample application has 12 tasks (i.e., T_1 to T_{12}).

The major way to define an application is to use the ORBIT API. ORBIT provides the application developer an API, using Java annotations [Oracle,]. By using this API, an application developer implements the application pipeline as a Java class specifying each task in the pipeline as a *field* and uses ORBIT-provided annotations to annotate each task. By annotations, the developer indicates which task is connected to another task(s) as well as which outputs data pins in the source task are connected to which input data pins in the destination task. For instance, a Java class generating the application pipeline in Fig. 2.7 can simply be implemented as shown in Fig.. 2.8, where $Task_i$ is an algorithm in the ORBIT library, the $param_i$s specify the input and output parameters for each task including the input, output data and data sizes (number of samples) and other algorithms' specific parameters, e.g., threshold, window size and etc. The $@Next$ annotation is defined by ORBIT API and used by application developer to connect the tasks and form the pipeline. The annotations $@source$ and $@sink$ are used to indicate the source and sink

```
/** import ORBIT API **/
public class Sample_application_pipeline extends ORBIT_pipeline_model {
    @Source
    @Next{T_2, T_3}
    private Task T_1 = new Task_1(param_1,param_2,...,param_N);
    @Next{T_4, T_5{2}, T_6{1}}
    private Task T_2 = new Task_2(param_1,param_2, ...,param_N);
    @Next{T_7,T_8}
    private Task T_3 = new Task_3(param_1,param_2,...,param_N);
    . . .
    @Sink
    private Task T_10 = new Task_10(param_1,param_2,...,param_N);
}
```

Figure 2.8: Pseudo-code for generating an application pipeline

tasks in the pipeline.

A key advantage of the annotation-based ORBIT programming model is that the developers use the advanced features of Java supported by Android and take advantage of the ease of use of Java language to set up the application pipeline without being burdened with error-prone embedded programming using low-level languages.

2.6.2 Data Processing Library

ORBIT provides a library of data processing algorithms ranging from common learning algorithms and utilities (e.g., classification, regression, clustering, filtering, and dimensionality reduction), to primitives like gradient decent optimizations. Using these well tested functions and provided APIs, developers can quickly construct sensing applications by simply connecting different building blocks via the ORBIT application pipeline model. This library has two main design objectives. Firstly, it is extensible so that developers can easily add more algorithms or port legacy signal pro-

cessing libraries. Secondly, it is designed to be resource-friendly with smartphone and extBoard (if utilized by the application). Several algorithms are implemented in Java while others are written in C++ and connected with the rest of ORBIT components via a Java Native Interface (JNI) bridge.

A key challenge in the design of ORBIT programming library is that many ORBIT applications have stringent requirements on timing/overhead. ORBIT library includes two mechanisms to optimize resource usage while providing programming flexibility at the same time, namely adaptive delay/quality trade-off and data partitioning via multi-threading. These mechanisms allow programmers to develop *resource-friendly* applications on the smartphone platforms.

2.6.2.1 Adaptive Delay/Quality Trade-off

The goal of this feature is to shorten the execution time of many tasks without substantially impeding the quality of their output. ORBIT achieves this by taking advantage of a property common to many algorithms. That is, many algorithms are iterative and based on an optimization function. The most commonly used methods to solve optimization problems, including the gradient descent method and Newton's method, are implemented as low-level primitives in the ORBIT library. Gradient descent is an iterative process moving in the direction of the negative derivative in each step (or iteration) to decrease the loss. Once the loss is less than a threshold, the algorithm stops. Similarly, Newton's method uses the second derivative to take a better route. Thus, a task that goes through more iterations to find the optimum solution for an objective function experiences a longer execution time, consequently causing the application to consume more energy on the smartphone. One way to shorten this latency and thus decrease the energy consumption is to simply stop the algorithm earlier, e.g., when the solution at step t is *satisfactory*. This approach is motivated by the principles of anytime algorithms [Zilberstein, 1996]. This *early-stopping* mechanism for these iterative-optimization tasks in ORBIT is controlled by three parameters: *stepSize, numOfItera-*

tions, samplingFraction, where *samplingFraction* is the fraction of the total data sampled in each iteration to compute the gradient direction. In the ORBIT library, these parameters are used as input parameters to the quality controller for each task while still satisfying the quality level of the entire application pipeline.

2.6.2.2 Data Partitioning via Multi-threading

One of the key advantages of the smartphone, in comparison to the mote-class platforms, is the availability of high-speed multi-core processors. Many smartphones today have two or more cores. For instance, Moto G costs less than $110 and has 4 cores. However, in spite of the availability of multi-core CPUs, multi-thread programming remains challenging. ORBIT can automatically partition long-running and compute-intensive tasks into different threads and run them on different cores. This allows users to focus on the domain specific aspects when designing the task structure for their sensing applications.

There are two different approaches to transforming an application into multiple threads. First, we can schedule different tasks of the application to execute on a pool of worker threads. In particular, ORBIT can parse the task structure and schedule tasks to different threads accordingly. However, many embedded applications contain a small number of "bottleneck" tasks in the signal processing pipeline, whose execution time dominates the total latency. As a result, such a task-level multi-threading strategy would not significantly reduce the end-to-end latency. ORBIT adopts a data-driven multi-threading approach to partition these tasks. We now use the *matrix-vector multiplication* operation as an example to illustrate this approach.

Many signal processing algorithms (e.g., various transforms and compressive sampling) are based on matrix multiplication. The output $\mathbf{y}$ is the matrix multiplication expressed as $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A} \in \mathbb{Z}^{m \times l}$ is the computation to be applied on the input $\mathbf{x} \in \mathbb{Z}^{l \times 1}$. Suppose matrix $\mathbf{A}$

is evenly split into sub matrices, i.e., $\mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_K]$, where $\mathbf{A}_k \in \mathbb{Z}^{m/k \times l}$. The kth sub-task computes $\mathbf{y}_k = \mathbf{A}_k\mathbf{x}$, and the final result is $\mathbf{y} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k]$. The kth sub-task also performs matrix-vector multiplication. ORBIT picks the value of $\mathbf{k}$ based on the number of cores available on the phone (which can be queried through an Android API). ORBIT creates the computation threads on-the-fly and assigns the maximum priority to them to ensure they will not compete for resources with other threads running on the device. In this manner, ORBIT splits all matrix-based signal processing tasks assigned to the smartphone.

A number of signal processing algorithms based on matrix operations can benefit from ORBIT's data partitioning scheme. Examples include Singular Value Decomposition (SVD), Eigenvalue Decomposition, Principal Component Analysis (PCA), mean and average. These fundamental algorithms are often used in the design of other more advanced algorithms. Since extBoard does not support multi-threading, these versions are implemented in C++ without the use of any matrix libraries.

A key design consideration of multi-threading is to minimize the overhead of inter-thread communication. In ORBIT, the matrices are passed to the threads by reference and each thread computes the partial and *non-overlapping (disjoint) part* of the result. In other words, different threads access the same data structure but disjoint parts of it. For example, thread $\mathbf{k}$ computes $\mathbf{y_k} = \mathbf{A_k}\mathbf{x}$, and sub-matrices $\mathbf{y_k}$ are not overlapping. Matrix $\mathbf{y} = [\mathbf{y_1}, \mathbf{y_2}, \ldots \mathbf{y_k}]$ is also accessed by the main thread similarly without conflicts or memory copy between threads. The avoidance of inter-thread communication in ORBIT is important for data-intensive tasks that deal with large matrices.

2.6.3 Task Partitioning and Energy Management

A key design objective of ORBIT is to provide an energy efficient smartphone-based platform. For this purpose, ORBIT adopts a task partitioning framework that exploits the heterogeneity in power consumption and latency profiles of different tiers. The task partitioning algorithm minimizes system energy consumption while meeting the processing deadlines of sensing applications. In addition, to reduce application delays, ORBIT implements a data partitioning scheme that decomposes matrix-based computation into multiple threads which are scheduled to execute on different CPU cores.

2.6.3.1 Power Management Model

From the key observations obtained from the measurement-based study in Section 2.5, ORBIT employs different power management strategies for different tiers. Specifically, the extBoard operates in a duty cycle where it remains active for T_a seconds and sleeps for T_s seconds in a cycle. During the active period, the extBoard samples the sensors at constant rates. The time duration for sampling a signal segment is referred to as *sampling duration*, and denoted as T_d. The active period contains multiple sampling periods. A signal segment collected during the current sampling period will be processed by the ORBIT application (e.g., the one shown in Fig. 2.7) in the next sampling period. The values of T_a, T_s, and T_d are determined based on the expected system lifetime and timeliness requirements of the sensing application. Moreover, the sampling and processing on the extBoard are often subjected to stringent delay bounds. Modern microprocessors also offer low power sleep states with wake on interrupt which can be utilized to further reduce the extBoard power consumption during the sampling period. Different from extBoard, the smartphone adopts an on-demand sleep strategy in which it remains asleep unless activated by extBoard or by the

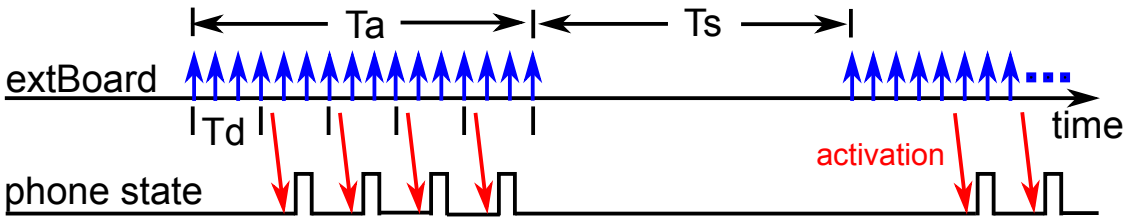


Figure 2.9: Power management scheme.

cloud messages. Fig. 2.9 illustrates the extBoard's duty cycle and the smartphone on-demand sleep schedule.

2.6.3.2 Execution Time Profiler

The extBoard and smartphone power profiles are unlikely to substantially change during the lifetime of the application. However, the latency profile of a task may contain errors and be subject to change after deployment, as shown in the Fig. 2.4 example. To address this issue, ORBIT continuously measures the latency of each task at runtime and periodically re-runs the task partitioner to update the task partitioning scheme. Specifically, we designed an Execution Time Profiler that can build the statistical latency models for all tasks based on the run-time measurements. It measures the execution time of each task by using the system time before and after execution of the task. It also maintains a Gaussian distribution model for each task's execution time, $T_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$. The parameters of this distribution are updated by each new measurement t as : $\mu_i' = \mu_i + \frac{1}{n}.(t - \mu_i)$ and $\sigma_i'^2 = \frac{1}{n}((n-1)\sigma_i^2 + (t-\mu_i)(t-\mu_i'))$. Based on these models, the percentiles with a high rank are used to set the execution times (i.e., t_i^{p}, t_i^{b}, and t_i^{c}). Under this approach, ORBIT can achieve optimal partitioning solution while meeting the timing requirements statistically.

2.6.3.3 Partitioning with Sequential Execution

As discussed in Section 2.6.3.1, the extBoard has a fixed duty cycle and hence consumes relatively constant energy. Therefore, ORBIT aims to minimize the total energy consumption of smartphone, subject to the processing delay upper bound for each tier. Consider a sensing application consisting of n tasks (denoted by $T_1, \ldots, T_n$), with an execution pipeline expressed as a sequential set of tasks: $\mathbb{T} = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$. Let I_i denote the execution tier of T_i, where: $I_i \in \{(1,0,0),(0,1,0),(0,0,1)\}$ represent the extBoard, smartphone, and cloud, respectively. Let τ_b, τ_p, τ_c, τ_A denote the execution times of the extBoard, smartphone, cloud, and the end-to-end delay of the whole application (or the delay-critical portion of the application), respectively, in a sampling period. We now formulate the task partitioning problem for sequential execution. The case of branching execution is discussed in a technical report [Moazzami et al., 2013].

Task Partitioning Problem. *For the sequential execution $\mathbb{T} = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, the Task Partitioner finds an execution assignment set $S = \{I_1, I_2, \ldots, I_n\}$ to minimize the total smartphone energy consumption in a sampling duration (denoted by E) subject to $\tau_b \leq D_b$, $\tau_p \leq D_p$, $\tau_c \leq D_c$, and $\tau_A \leq D_A$.*

The processing delay upper bounds D_b, D_p, D_c, and D_A are typically set according to the timeliness requirements of the application, e.g., the constant rate of sensor sampling, the time period to detect a moving object before it moves away, etc. As the sensor sampling and timestamping introduce little overhead (cf. Section 2.6.4.2), it is safe to set D_b to a value that is slightly smaller than the sampling period. It is shown in [Newton et al., 2009] and [Cuervo et al., 2010b] that this partitioning problem is modeled as an integer linear program (ILP) that minimizes a linear combination of network bandwidth and CPU consumption subject to the upper bounds for these resources. It is important to note that under the conventional ILP partitioning, the model only takes

the execution time latency (i.e., CPU consumption) and data copy latency between tiers (e.g., network bandwidth) into account. In contrast, ORBIT extends this model by adding two additional terms to the partitioning model. These terms are wake-up and tail time of smartphone and the (instant) power consumption of each tier. Also, with the help of the execution time profiler, ORBIT considers one more factor, the uncertainty of execution times. Thus, ORBIT provides a more realistic partitioning model.

We now derive E and the delays (τ_A, τ_b, τ_c, and τ_p). We first define the following notation. The execution times of task T_i on the extBoard, smartphone and cloud are denoted by t_i^{p}, t_i^{b} and t_i^{c}, respectively. Let P denote the power consumption, where the superscripts 'p', 'b', and 'c' represent smartphone, extBoard, and cloud; and the subscripts 'a' and 's' represent active power and sleep power of the smartphone and the extBoard. Denote $t_{\mathrm{b}\leftrightarrow\mathrm{p}}$ the latency of downloading/uploading a data unit from/to the phone to/from the extBoard, $t_{\mathrm{p}\leftrightarrow\mathrm{c}}$ the latency of downloading/uploading a data unit from/to the phone to/from the cloud, J_i the number of input pins of T_i, and $l_{i,j}$ the signal length of the jth input pin of T_i.

We now analyze the energy consumption and processing delay of an application in a sampling period. Note that we only need to analyze the energy consumption of the smartphone. The reasons are twofold. First, the cloud's energy consumption does not fall into the system's total energy consumption. Second, as the extBoard keeps active to continually sample sensors or activate the smartphone, its power consumption is fixed.

(1) Processing energy consumption and delay: Let E_1 and τ_1 denote the smartphone energy and delay in processing the signal collected during a sampling duration. Analysis shows

$$E_1 = \sum_{i=1}^{n} I_i \cdot \begin{pmatrix} (P_{\mathrm{a}}^{\mathrm{b}} + P_{\mathrm{s}}^{\mathrm{p}}) t_i^{\mathrm{b}} \\ (P_{\mathrm{s}}^{\mathrm{b}} + P_{\mathrm{a}}^{\mathrm{p}}) t_i^{\mathrm{p}} \\ (P_{\mathrm{s}}^{\mathrm{b}} + P_{\mathrm{s}}^{\mathrm{p}}) t_i^{\mathrm{c}} \end{pmatrix} + d(I_i, I_{i-1}) \cdot \frac{E_w + E_T}{2},$$

$$\tau_1 = \sum_{i=1}^{n} I_i \cdot \begin{pmatrix} t_i^{\mathrm{b}} \\ t_i^{\mathrm{p}} \\ t_i^{\mathrm{c}} \end{pmatrix} + d(I_i, I_{i-1}) \cdot \frac{T_w + T_T}{2},$$

where E_w, T_w, E_T and T_W are the energy consumed and the time spent during the tail and wake-up phases respectfully. The function $d(I_i, I_{i-1})$ accounts for the data-copy overhead between the tiers by indicating the distance between the positions of '1' in I_i and I_{i-1}. For instance, when $I_i = I_{i-1} = (1, 0, 0)$, $d(I_i, I_{i-1}) = 0$; when $I_i = (1, 0, 0)$ and $I_{i-1} = (0, 1, 0)$, $d(I_i, I_{i-1}) = 1$. Since the number of tiers are three, the maximum distance is two $d_{max}(I_i, I_{i-1}) = 2$ and that is when a task is assigned to the cloud (e.g., $I_i = (0, 0, 1)$) while its previous task is assigned to the extBoard (e.g., $I_{i-1} = (1, 0, 0)$), or vice-versa. In this case, the data has to be transferred between these two tiers through smartphone (there is not direct communication between extBoard and the cloud server). Moreover, we obtain the execution times of the extBoard, smartphone and the cloud portion of application as following:

$$\tau_b = \sum_{i=1}^{n} I_i \cdot \begin{pmatrix} t_i^{\mathrm{b}} \\ 0 \\ 0 \end{pmatrix}, \tau_p = \sum_{i=1}^{n} I_i \cdot \begin{pmatrix} 0 \\ t_i^{\mathrm{c}} \\ 0 \end{pmatrix}, \tau_c = \sum_{i=1}^{n} I_i \cdot \begin{pmatrix} 0 \\ 0 \\ t_i^{\mathrm{c}} \end{pmatrix}$$

(2) Overhead of phone state transitions and cross-tier data copy: Let E_2 and τ_2 denote the energy consumption and the delay for copying data. We define a function $s(i, j)$ based on the application pipeline which returns the ID of the source task connected with the T_i's jth input parameter. If the tasks T_i and $T_{s(i,j)}$ are executed at different tiers, the jth input data parameter of T_i needs to be copied between the consecutive tiers causing smartphone energy consumption of $P_{\mathrm{a}}^{\mathrm{p}} t_{\mathrm{c}} l_{i,j}$ and extBoard processing delay of $t_{\mathrm{c}} l_{i,j}$. Thus,

$$E_2 = \sum_{i=1}^{n} \sum_{j=1}^{J_i} d(I_i, I_{s(i,j)}) P_{\mathrm{a}}^{\mathrm{p}} t_{\mathrm{c}} l_{i,j}$$

$$\tau_2 = \sum_{i=1}^{n} \sum_{j=1}^{J_i} d(I_i, I_{s(i,j)}) t_{\mathrm{c}} l_{i,j}.$$

Therefore, the total smartphone energy consumption and the delay for processing the sensor data collected in a sampling period are $E = E_1 + E_2$ and $\tau_A = \tau_1 + \tau_2$. Note that E does not include the sleep energy consumption of the smartphone from the end of the current execution cycle to the beginning of the next cycle when the new sensor data become available. However, as the Task Partitioner will fully utilize the allowed processing time D to reduce the smartphone energy consumption, the time duration of an execution cycle will be close to the sampling period if D is close to the sampling period. Therefore, the sleep energy consumption of the smartphone during the gap is negligible. Based on the above delay and energy models, the task partitioning problem is a constrained non-linear optimization problem. The nonlinearity comes from the formula of E and τ. ORBIT uses brute-force search to solve the problem. As the number of tasks in a sensing application is typically small, our measurements in Section 3.7 show that the brute-force search introduces little overhead even if the Task Partitioner is executed periodically by the smartphone

(c.f., Section 2.6.3.4). For instance, its execution time is less than 10 ms on a Galaxy Nexus for up to 20 tasks.

2.6.3.4 Partitioning with Branches

While we focus on sequential execution in the last section, real applications can contain branches in their execution flow. To discuss our approach to partitioning tasks containing branches, we continue to use the running example shown in Fig. 2.7. Different from sequential tasks, a key challenge is a task partitioning solution that is optimal for all branches may not exist. As an example, consider the part of Fig. 2.7, which includes T_1, T_2, and T_3 only. Suppose we run the task partitioning algorithm for the two execution paths, i.e., $T_1 \rightarrow T_2$ and $T_1 \rightarrow T_3$. These two solutions can be conflicting because T_1 may be assigned to different tiers (smartphone and extBoard) in each solution. ORBIT adopts a priority-based approach to resolve the potential conflicts. In the execution tree of Fig. 2.7, there are six paths from the root node to all leaf nodes. We assign an integer priority to each path, where a smaller number means a higher priority. As each leaf node is associated with a unique path from the root node, the priorities can also be associated with the leaf nodes. A higher priority means that the corresponding path will be executed with higher probability. The priorities can be assigned by the developer or randomly set by default. In our approach, we run the task partitioning algorithm discussed in Section 2.6.3.3 for each path, in the order of increasing integer priority. For instance, we run the task partitioning algorithm over the path with the highest priority (i.e., $T_1 \rightarrow T_2 \rightarrow T_6$), yielding solution S_1. We then choose the path with the second highest priority (i.e., $T_1 \rightarrow T_2 \rightarrow T_5 \rightarrow T_9$). As T_1 and T_2 have been included in S_1, we run the task partitioning algorithm for the residual path (i.e., $T_5 \rightarrow T_9$) only with the assignment of T_2 in S_1. We apply this procedure to all other paths. At run time, the Task Controller executes the tasks according to the assignment set. For instance, in Fig. 2.7, if it decides to execute

T_5 according to the result of T_2, it will dispatch T_2 according to S_2.

2.6.4 Task Controllers

Task Controllers (TCs) on the smartphone, the extBoard, and the cloud execute the entire sensing application according to the assignment computed by the Task Partitioner. Fig. 2.2 shows the interaction of the TCs with other components in ORBIT.

2.6.4.1 Smartphone Task Controller

The smartphone TC is designed as an Android background service, which manipulates the execution of the tasks and communicates with the extBoard and the cloud. When the ORBIT application is launched, the smartphone TC creates the instances of the tasks in $\mathbb{T}$, and allocates the buffers for all inputs and outputs. After this initialization phase, the TC checks the partitioning assignments and begins execution of the first task. When the smartphone is not executing a task, it switches to the sleep state to conserve energy. When task execution needs to return to the smartphone, a notification message is sent waking the smartphone and activating its TC to execute the next tasks assigned to it. The smartphone TC also continuously updates the task meta information (e.g., execution times) as well as branch priorities.

Our measurement study shows that the smartphone consumes considerable energy during wake-up and tail phases (cf. Section 2.5.2). We optimize the design of TC to start a task as soon as the smartphone wakes up or to let the smartphone sleep as soon as no more tasks need to run. After the TC executes a task T on the smartphone, it checks if there is any task assigned to the other two tiers that takes T's output as input. If there are, TC will send T's output data to the other tier using ORBIT's messaging protocol [Moazzami et al., 2013]). This allows the other tiers to run the tasks with input data from smartphone without re-activating it, avoiding extra wake-up and tail energy

consumption. However, a side effect of this design is that, if the application has branches the data transmitted to another tier may not be used. However, typical signal processing pipelines likely contain a limited number of branches.

2.6.4.2 extBoard and Cloud Task Controllers

The extBoard TC continually checks for the arrival of messages from the smartphone. When it receives a *start task execution* message from the smartphone, it begins executing the first task in its assignment. In the case of starting a sampling task, the extBoard creates a periodic timer to control the sampling. The timer interrupt handling routine reads a sensor sample from the ADC, time-stamps it, and then inserts it into a circular buffer. This process involves only a few instructions, and is optimized to reduce the interrupt handling delay. Once the sampling task has obtained the number of samples specified in its input parameter, task execution continues with the task following the sampling task according to the execution tree $\mathbb{T}$.

The cloud TC is implemented as a Linux daemon that checks for the arrival messages from one or multiple smartphones. There are two types of tasks running on the cloud; tasks that are computationally intensive that are assigned by the Task Partitioner and the tasks that take input data from multiple ORBIT nodes. Upon the completion of task T in the cloud, the cloud TC sends T's output to all the smartphones that require the output. If any of the smartphones are in the sleep mode, they wake up when a cloud message is received. The cloud TC, like the smartphone TC, continuously updates the task meta information (e.g., execution times) as well as branch priorities. This ensures fresh meta information is used for the task partitioning.

2.7 Microbenchmark

In this section, we evaluate overall memory and CPU usage of ORBIT as well as the overhead introduced by online task partitioning. We also evaluate the effect of multi-threading on reducing the task processing delays.

CPU and memory footprint on smartphone: We measure the CPU and memory footprints of ORBIT. We use the Android utility application, System Monitor, to measure the CPU and memory usages. We select different applications that run with ORBIT. ORBIT runs as an app and its CPU utilization may vary based on the smartphone hardware, Android version and other apps running on the smartphone. We measure the CPU footprint of ORBIT by the increased CPU utilization when it runs tasks. Our measurements show that ORBIT's CPU footprint ranges from 10% up to 15%. The memory usage is about 22.5 MB during silence, but reaching 33.8 MB for a sensing application as heap space is dynamically allocated for the processing tasks. The total size of the ORBIT binary is only 2.84 MB.

Delay/quality trade-off: In section 2.6.2.1 we discussed how algorithms are tuned for desirable trade-offs between quality and delay. Fig. 2.10 shows the convergence of the Gradient Descent algorithm for different step sizes r and number of iterations. As it is expected and is illustrated in the figure the gradient value decreases as the number of iterations increases until finally converges to the solution. Larger step sizes result in the gradient converging faster. However the rate of decrease slows after a certain iteration for each step size, meaning the task does not benefit from more iterations. Thus, gradient descent can often find a *good enough* solution in fewer iterations than the number of iterations provided as an input parameter, allowing ORBIT to stop it earlier without loosing a significant accuracy. Examples of algorithms that can benefit from this feature are SVM, linear regression and K-mean clustering. This feature not only provides insights for

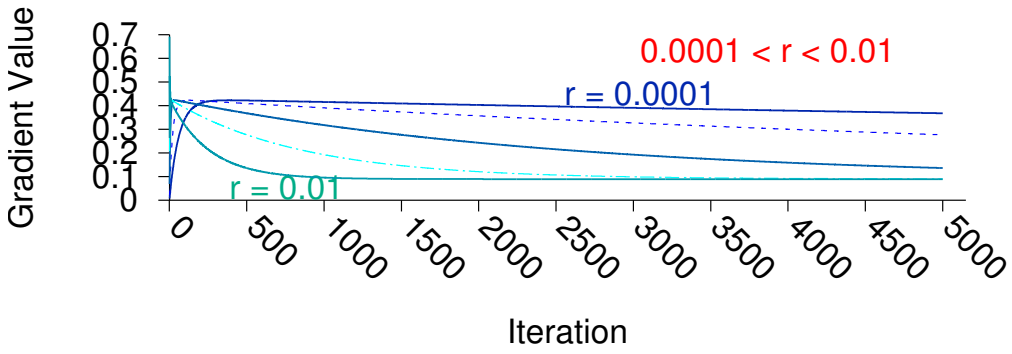
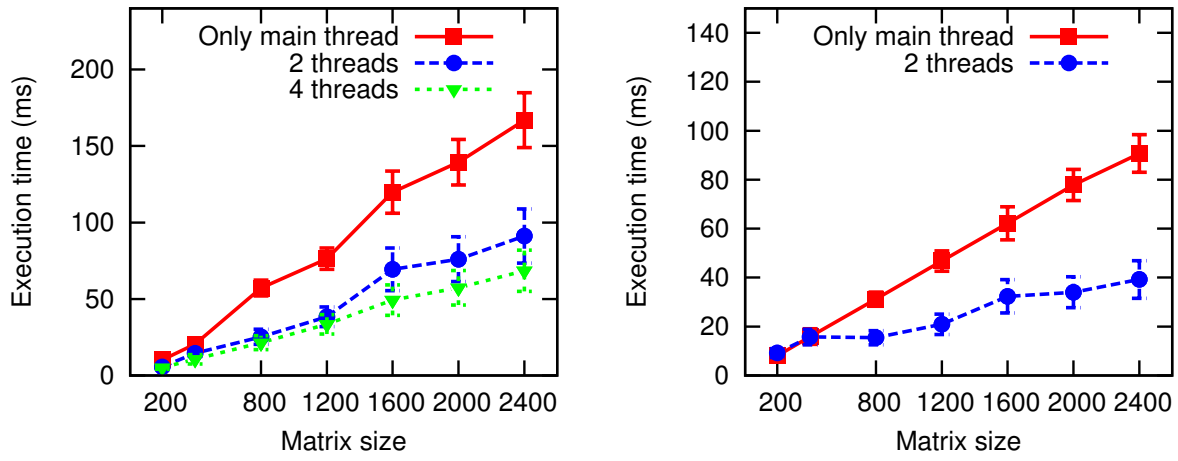


Figure 2.10: Delay/quality trade-off (r = step size)

choosing better parameter values for each task in the application pipeline, but also gives ORBIT the power and a systematic mechanism to terminate the tasks while still maintaining the results within an expected accuracy range.

Effect of data partitioning and multi-threading: As discussed in Section 2.6.2.2, smartphone TC can partition compute-intensive tasks into multiple threads to reduce the processing delays. Fig. 2.11 shows the performance gain of a matrix vector multiplication task, $\mathbf{y} = \mathbf{Ax}$, on two different smartphones, Moto-G with a quad-core processor and Galaxy Nexus with a dual-core processor. In this example, vector $\mathbf{x} \in \mathbb{Z}^{l \times 1}$ is the input signal and matrix $\mathbf{A} \in \mathbb{Z}^{m \times l}$ is the computation matrix. l has a fixed value of 2000 data samples and m varies for different operations (the horizontal axis in the figure). Larger values of m indicate more data-intense computation. The results show that the computation delay reduces by 44.7%, on average, for Mote-G and reduces by 36.2% for Galaxy Nexus when the task is partitioned into 2 threads. It also reduces by 56.1% for Mote-G when the computation is partitioned into 4 threads.

As we can see from the figure, multi-threading reduces the computation delay more for larger matrices (more data-intensive computation) that agrees with our design objectives. Another important result from this figure is that 4 threads in Moto-G does not provide significant improvement over 2 threads. This is because, once the computation is partitioned into 2 threads the problem



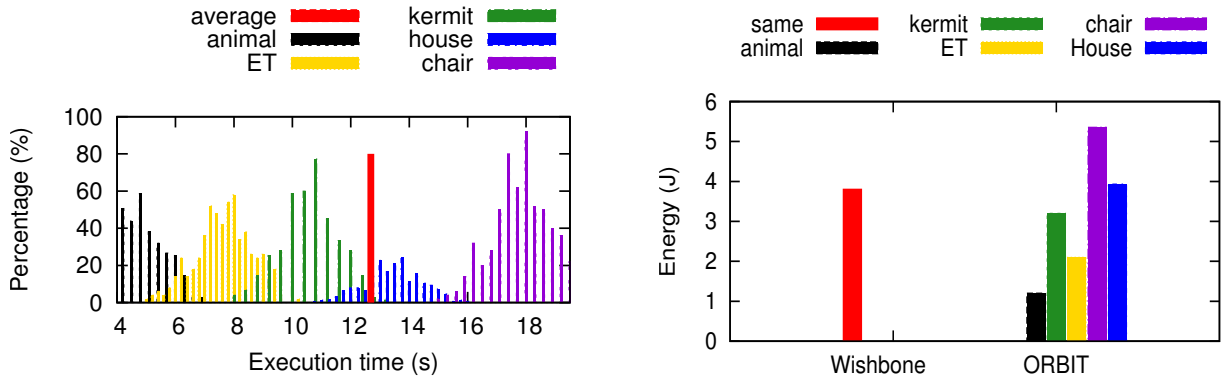
(a) Moto-G (4 cores)

(b) Galaxy Nexus (2 cores)

Figure 2.11: Smartphone multi-threading reduces processing delay of compute-intensive tasks.

size is reduced by half. Consequently when each thread is further split into 2 new threads, it only affects a smaller problem and thus the reduction in computation delay is smaller. This agrees with the intuition that multi-threading provides less improvement for smaller problems.

Effect of data dependency: A salient feature of ORBIT is that it takes input data size and input data content into account in modeling the task energy consumption and partitioning. In contrast, in conventional task modeling and partitioning schemes, the time latency is measured offline and the average value is often assumed as the time latency without considering the observed variance in the execution time. However, our measurement study shows that the execution time can vary significantly for a data-dependant algorithm with different input sizes and input content. We now use several examples to illustrate the effect of data dependency on the system energy consumption. Fig. 2.12a shows the distribution of the execution time for the SIFT algorithm for input images with different dimensions and number of SIFT features. Fig. 2.12b shows the difference between the energy consumption estimation of SIFT algorithm under the Wishbone approach and the approach adopted by ORBIT. Since Wishbone does not consider the differences between input data, the



(a) Execution time distribution of SIFT

(b) Energy consumption of SIFT for different images

Figure 2.12: The data-dependant algorithms.

average value of offline measurements will be used as the execution time. Therefore, when the execution time of SIFT for an image is close to the average value, e.g., for the house image, the energy estimated by both approaches are similar. However when the execution time of the image is less than the average, e.g., for the ET, kermit and the animal images, the estimated energy by ORBIT outperforms Wishbone. On the other hand, if the execution time is longer than the average execution time, e.g., the chair image, although the energy estimated by ORBIT is larger than Wishbone, ORBIT provides a closer estimation to the true value. Thus, ORBIT provides a more realistic approach to model the execution times and the energy consumption of data-intensive algorithms.

2.8 Case Studies

To demonstrate the expressivity of ORBIT application scripting as well as the generality and flexibility of ORBIT as a platform, we have prototyped three different embedded sensing yet data-intensive applications. (cf. Table 2.1). Each application demonstrates different facets of ORBIT varying the number of tasks in the task-structure, the use of different sensors, the number of tiers

the application is partitioned between, and the data fidelity requirement of the application. Our goal is to demonstrate the capabilities and effectiveness of the platform rather than present novel applications.

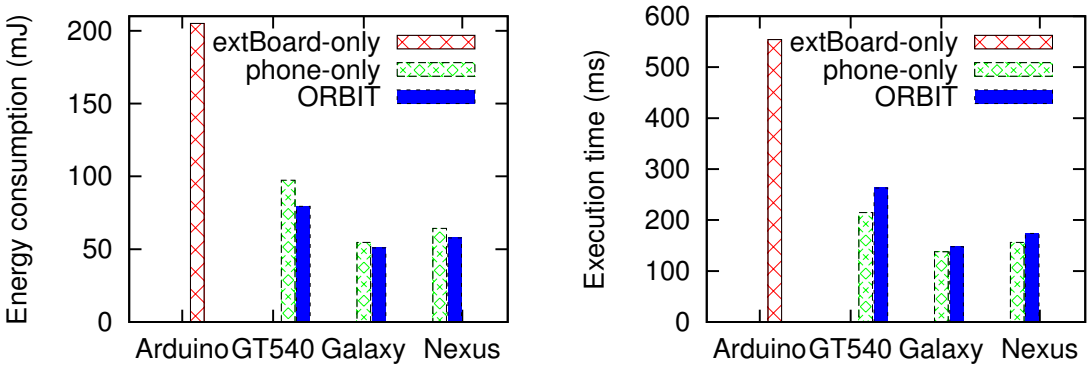
Table 2.1: ORBIT based applications.

Application ←	Robotic Sensing	Event Timing	Multi-Camera 3D Reconstruction
Script Length	35	27	20
Number of Tasks	11	7	10
Sensors	IR, Camera, Ultrasound	GPS, Geophone	Camera, GPS
Tiers	extBoard, smartphone	extBoard, smartphone	extBoard, smartphone, cloud
Data Fidelity	5fps	100Hz	640*480px

2.8.1 Robotic Sensing

We choose a robotic sensing application for our first case study. In this case study the application estimates the presence, the distance and the direction of an object approaching using an IR and a Sonar sensor attached to the robot as well as the smartphone's built-in camera (cf. Fig. 2.1b). Once an object is detected by IR sensor, its distance is estimated using the sonar sensor. If within a specified range, the extBoard sends a command to the smartphone, waking up the smartphone, and activating the camera.

Once the system captures the image, it converts the image to grayscale and the computes the threshold to separate the background and the foreground. Next, the system detects the objects and computes its bounding rectangle. As the object moves, its bounding rectangle changes. The direction and the velocity of this movement is determined by computing changes in the bounding rectangles. This information is then used to move the robot to track the object. We used a portion of the code from an open source soccer robot project [Object Tracking Robot,].



(a) Total energy consumption

(b) Total execution time

Figure 2.13: The results of various partition schemes

Compared to the event timing application, this application involves actuators in addition to the sensors. A few tasks are not allowed to be partitioned since they are directly sampling the sensors or actuating robot servos. Other processing tasks can be partitioned between tiers. The processing delay determines the maximum image capture frame rate which in turn determines the maximum speed of a moving object that can be tracked.

Effectiveness of Task Partitioner: We first evaluate the effectiveness of the task partitioning algorithm by comparing with the baselines. For this case study, our baselines are extBoard-only and smartphone-only. Fig. 2.13 shows the energy consumption and the application execution time when the delay bound D is set to $0.4\,\mathrm{s}$. Fig. 2.13a and Fig. 2.13b plot the estimated total energy consumption and total execution time (i.e., smartphone + extBoard) of a ORBIT node in one execution cycle. As the extBoard is slow and power-inefficient for intensive computation, it cannot meet the delay bound and consumes the most energy. Our partitioning approach in ORBIT achieves the lowest energy consumption across different smartphones.

Impact of delay bound: We then evaluate the impact of the delay bound D on the task assignment

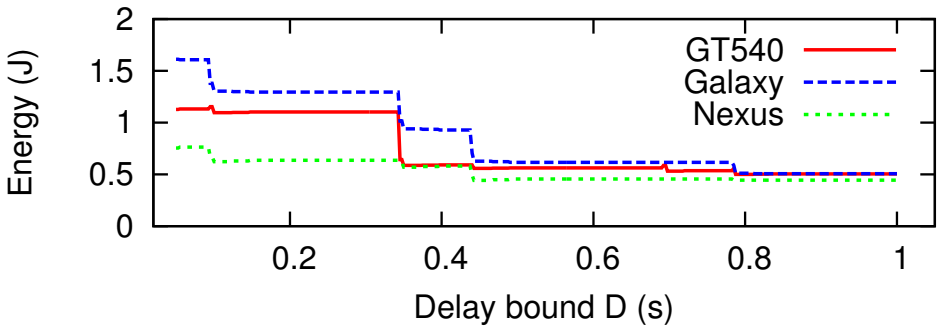


Figure 2.14: Impact of delay bound setting on the task assignment and total energy consumption.

and smartphone energy consumption. Assume at least n frames are required to detect the object and track its trajectory; the smartphone camera's angle of view is θ; and the object's distance to the robot is d. Also, let v indicates the speed of the object. Therefore, the time the objects takes to move away out of the camera's view is $t = \frac{2.d\tan\theta}{v}$. Thus, the system has to process n frames in t seconds. As a result, the upper bound D for processing one frame is $D = \frac{t}{n} = \frac{2.d\tan\theta}{n.v}$. This shows how the delay bound is inversely related to the object's velocity. For example, if the camera's angle of view is 18 degrees, the object's distance to the robot is 3 m, and 5 frames are required to detect and estimate an object's moving direction, for an object moving at 5 km/h the system must to process each frame in less than 0.28 seconds.

Fig. 2.14 shows the smartphone energy consumption versus D. We can see that the total energy consumption decreases with D. This is because with higher delay bound more tasks are assigned to the extBoard allowing smartphone sleep for a longer time. This is consistent with our analysis. In this case the smartphone only wakes up when an image must be captured. It then preprocesses the image and sends the results to the extBoard to detect the object.

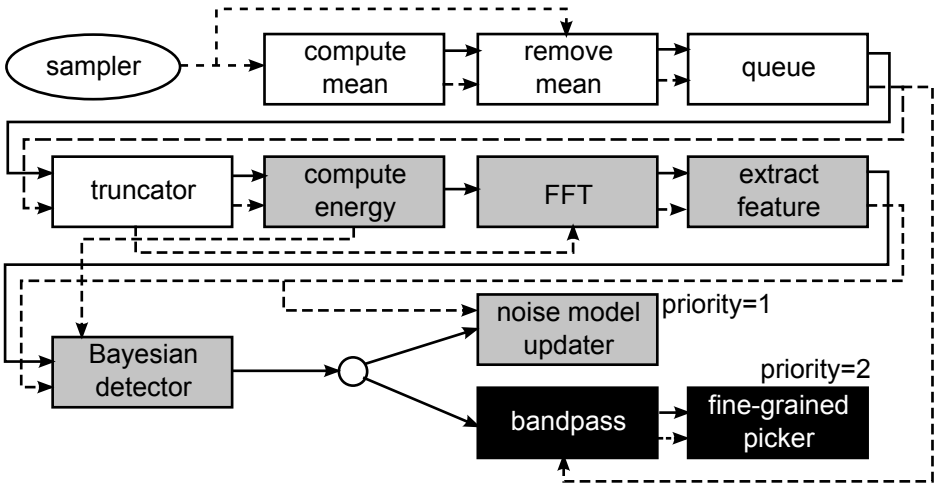


Figure 2.15: The block diagram of the seismic event timing application. The white blocks are pre-processing algorithms; the gray blocks are the earthquake detection algorithms; the black blocks are the P-phase estimation algorithms.

2.8.2 Event Timing

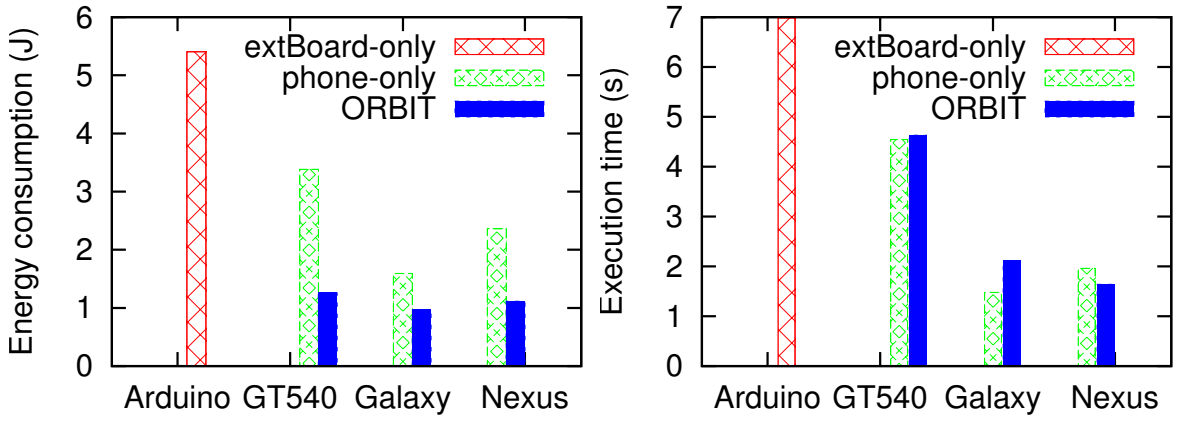
This application estimates the arrival time of an acoustic/seismic event. This is an building block of many acoustic/seismic monitoring applications such as distributed event timing [Liu et al., 2013] and source localization. Seismic event source localization requires events timed to sub millisecond precision and time synchronization between nodes to be within a few microseconds. Fig. 2.16 shows the application specification of this case study. The incoming signal is first pre-processed by mean removal and bandpass filtering. Wavelet transform is then applied to the filtered signal. Signal sparsity and coarse arrival time are computed based on the wavelet coefficients. This application requires a sampling rate of 100 Hz. In the context of early earthquake detection, the system must have a response time in the order of a few seconds. The following section presents the evaluation results.

Effectiveness of Task Partitioner: We first evaluate the effectiveness of the task partitioning algorithm presented in Section 2.6.3.3, by comparing the following partitioning approaches: extBoard only, phone-only, and greedy. The greedy approach assigns as many processing tasks to the

```
package com.my_sensing_app.pipeline;


import ORBIT.pipeline_model.*;
import ORBIT.io.*;
import ORBIT.generic_task;
import java.util.Vector;
...
public class Event_timing extends ORBIT_pipeline_model {
    @Source
    @fixed{"extBoard"}
    @Next{T_1, T_2{0}}
    private Task T_0 = new sampler("extBoard",1600);
    @Next{T_2{1}}
    private Task T_1 = new compute_mean(1600,1);
    @Next{T_3}
    private Task T_2 = new remove_mean(1600,1,1600);
    @Next{T_4}
    private Task T_3 = new filter("band_pass",1600,1600,1,6);
    @Next{T_5, T_6}
    private Task T_4 = new wavelet("haar",1600,1600,4);
    @Next{T_7}
    private Task T_5 = new compute_sparsity(1600,1);
    @Next{T_7}
    private Task T_6 = new coarse_picker(1600,1);
    @Sink
    private Task T_7 = new write_into_file("results.txt");
}
```

Figure 2.16: Application specification of event timing. The “sampler” is a special task running on the extBoard. Specific tasks with different parameters are defined. For example, the parameters “1600” and “1” indicate the number of input and/or output data samples for different tasks, the parameter “1,6” of the bandpass filter specifies the two corner frequencies; the parameter “4” of wavelet specifies the level of transform.



(a) Total energy consumption

(b) Total execution time

Figure 2.17: The results of various partition schemes.

extBoard as can be supported by the delay bound. Fig. 2.17 shows the task partitioning results of the partitioning approaches using a delay bound D of 1.8 s. The extBoard processing delay meets this bound except for the *extBoard-only* approach. Fig. 2.17a and Fig. 2.17b plot the estimated total energy consumption and total execution time (i.e., phone + extBoard) of a ORBIT node in one execution cycle under different partition approaches. As the extBoard is slow and power-inefficient for intensive computation, it cannot meet the delay bound and consumes the most energy. Our partitioning approach ("ORBIT") achieves the lowest energy consumption on the smartphones tested.

The impact of the delay bound D on the task assignment and smartphone energy consumption was next evaluated. The top portion of Fig. 2.18 shows the number of tasks assigned to the extBoard versus D. We can see that the Task Partitioner generally assigns more tasks to the extBoard for larger D. This is consistent with our analysis in Section 2.6.3.3. However, we can see a number of drops in the top portion of Fig. 2.18. For instance, when D increases from 1.37 s to 1.38 s, the number of extBoard tasks drops from 4 to 1. This is due to a compute-intensive task replacing the previous four lightweight tasks to increase the CPU utilization of extBoard and

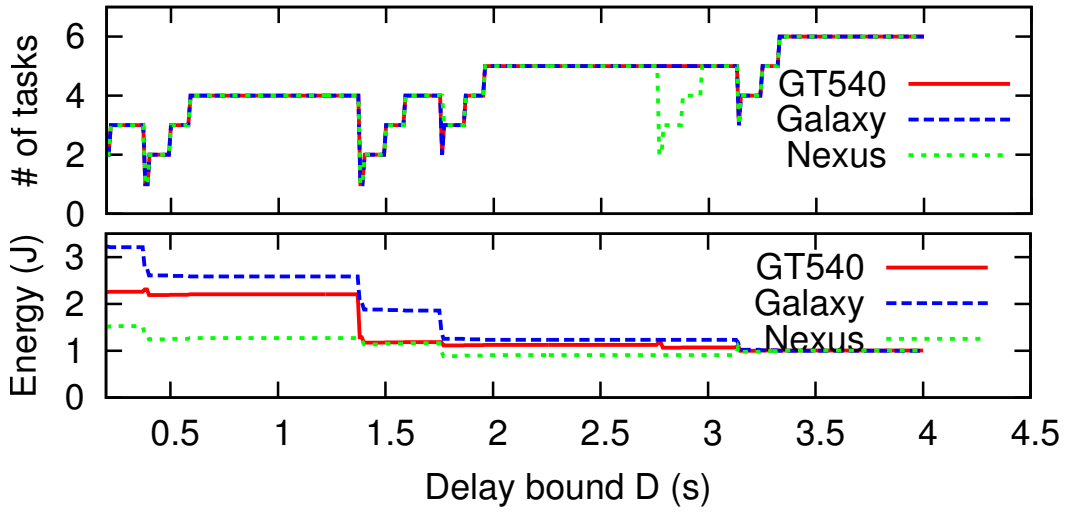
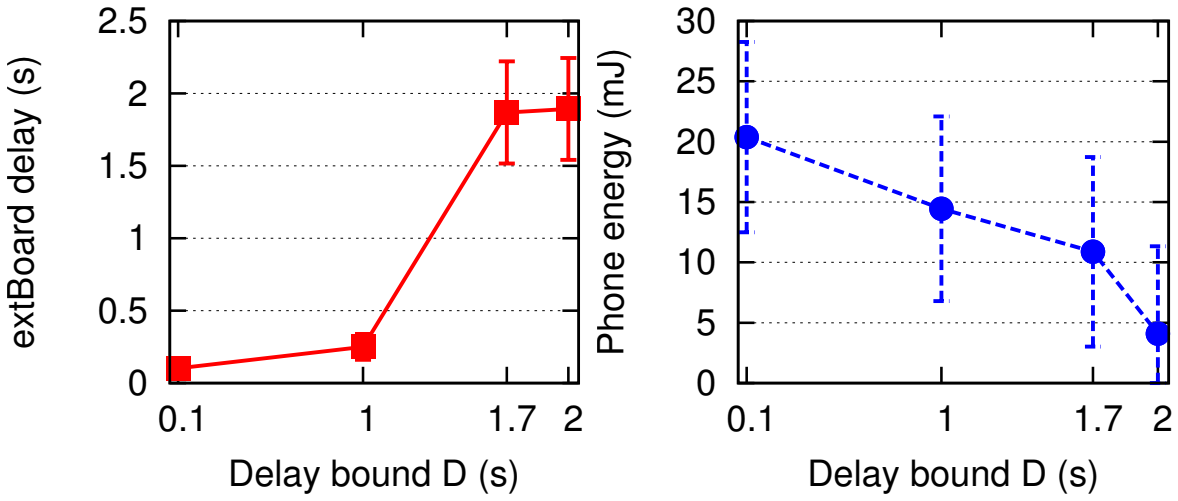


Figure 2.18: Impact of delay bound setting on the task assignment and total energy consumption. Top: The number of tasks assigned to the extBoard versus delay bound. Bottom: Total energy consumption versus delay bound.

reduce the smartphone energy consumption. The bottom portion of Fig. 2.18 shows the total energy consumption versus D. This shows the total energy consumption decreases with D, which is consistent with our analysis.

Measured execution time and energy consumption: Based on the obtained task partitioning results, we use a Nexus ORBIT node to run the application over real-time sensor readings. Fig. 2.19 plots the measured extBoard processing delay and the smartphone energy consumption versus the specified delay bound. Note the smartphone processing delay is less than 5 ms for all settings of delay bound. Therefore, the extBoard processing delay dominates. From Fig. 2.19a we can see that the specified delay bound is always met. Moreover, the extBoard processing delay increases with the delay bound, proving the effective utilization of the allowed extBoard CPU time. From Fig. 2.19b, the smartphone energy consumption decreases with the delay bound, which is consistent with our analysis.



(a) extBoard processing delay

(b) Phone energy consumption

Figure 2.19: The measured extBoard processing delay and smartphone energy consumption versus delay bound.

Duty cycle of extBoard and lifetime: Based on the measured energy consumption, we calculate the projected node lifetime over four D-cell batteries (capacity: 1.2×10^4 mAh) versus duty cycle of extBoard under various settings of delay bound. The results are plotted in Fig. 2.20. When the duty cycle is 100%, the projected lifetime is 5.8 days and when the duty cycle is 20%, the node can live for up to 2 months. As shown in Fig. 2.19b, the smartphone energy consumption is tens of millijoules, while the extBoard energy consumption is about one joule when duty cycle is 100%. Since the active powers of extBoard and smartphone are comparable (cf. Section 2.5), the extBoard energy consumption dominates when its duty cycle is large. In such cases, the major role of the smartphone is to help meet the tight delay bound, and the node lifetimes are similar for different delay bounds. However, when duty cycle is 20%, the lifetime can be extended by 18.4% if the delay bound increases from 0.1 s to 2.0 s. Nevertheless, with the help of the smartphone, the ORBIT node can meet tight delay bounds, which is critical to the success of many sensing applications that requires continuous sensor sampling.

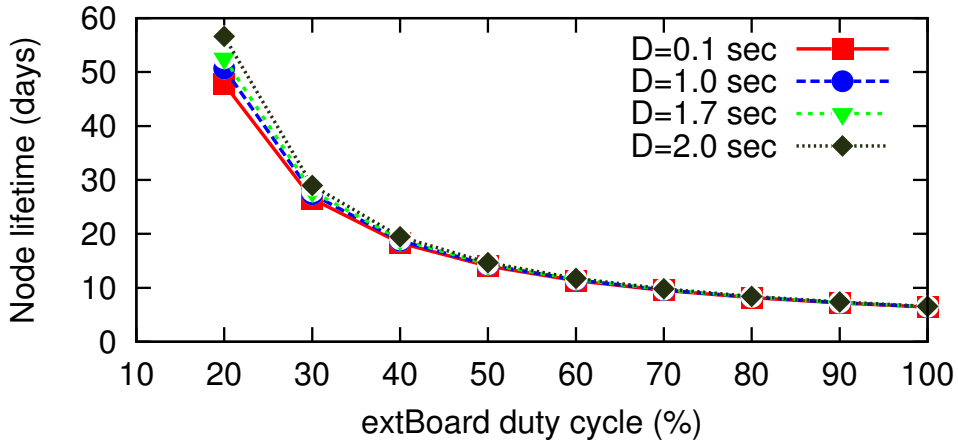


Figure 2.20: Projected lifetime vs. extBoard duty cycle.

Effect of branches: To evaluate the effect of branches, we integrate the event timing application in [Moazzami et al., 2013] with an event detection approach [Tan et al., 2010]. Fig. 2.15 shows the block diagram of the application. The sampling rate is 2.5 kHz, and the sampling duration is 40 milliseconds. In each cycle, the ORBIT node inserts the most recent 100 seismic samples into a *queue* with size of 1600. Next, the *truncator* task copies 100 samples at the middle of the *queue* to its output. A Bayesian event detection approach [Tan et al., 2010], which consists of multiple tasks, is applied to the output of the *truncator*. If the detector makes a positive decision, the node will run a primary-wave arrival time (i.e., P-phase) estimation algorithm [Sleeman and van Eck, 1999] based on all samples in the queue; otherwise, the node will use the input to the Bayesian detector to update the noise model used by the detector. Suppose the application is monitoring rare events (e.g., earthquakes), the execution path that branches to the *noise model updater* is assigned with higher priority since it occurs most frequently. In this case all tasks except the compute-intensive *bandpass* and *fine-grained picker* are assigned to the extBoard. Fig. 2.21 plots the trace of energy consumption of a tested ORBIT node in each sampling duration over time. In the first 7 seconds, the detector makes negative decisions and the node executes the branch to the *noise model updater*. From the 7th second, the detectors makes positive decisions for about 0.5 seconds and the extBoard

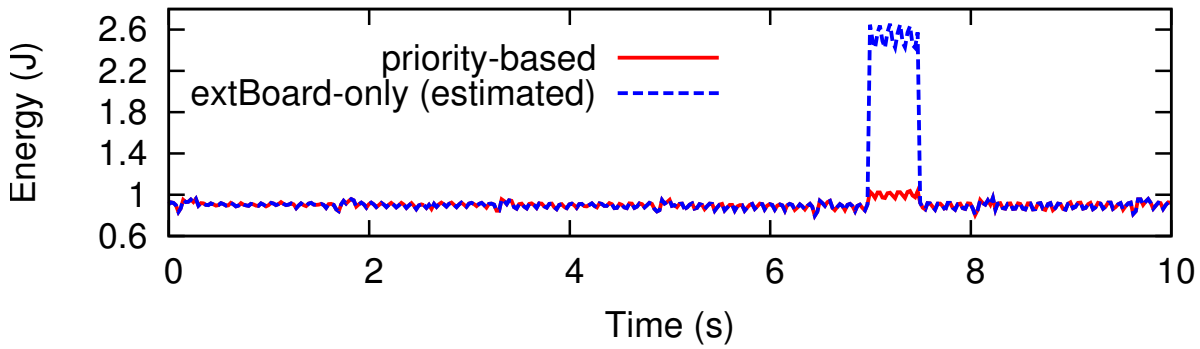


Figure 2.21: The energy consumption trace of a node.

activates the phone to execute the *bandpass* and *fine-grained picker*. Hence, we observe increased energy consumption. We compare our approach with the *extBoard-only* approach. Under this approach, the delay bound cannot be met when the detector makes positive decisions. Therefore, we can not directly run this approach on the node. Instead, we estimate the energy consumption based on the extBoard power model and task meta records. From Fig. 2.21, it can be seen that, when a seismic event occurs, the energy consumption of *extBoard-only* is significantly higher than our approach, primarily due to the long execution time of *fine-grained picker* on the extBoard.

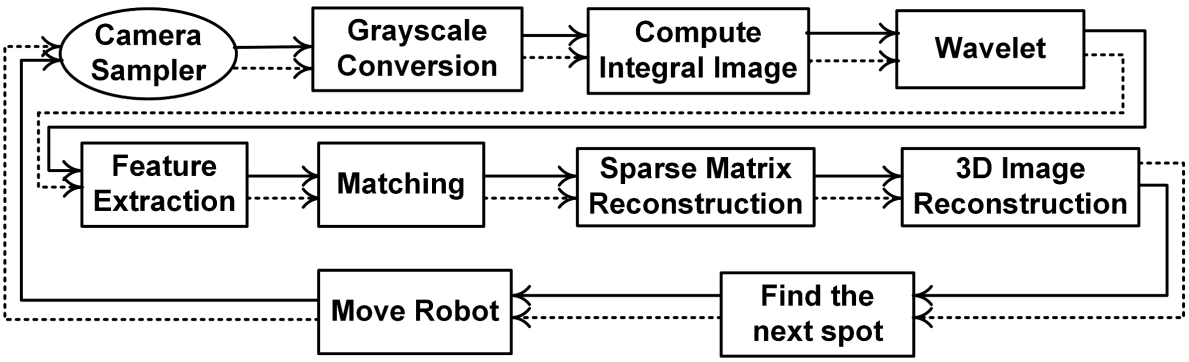
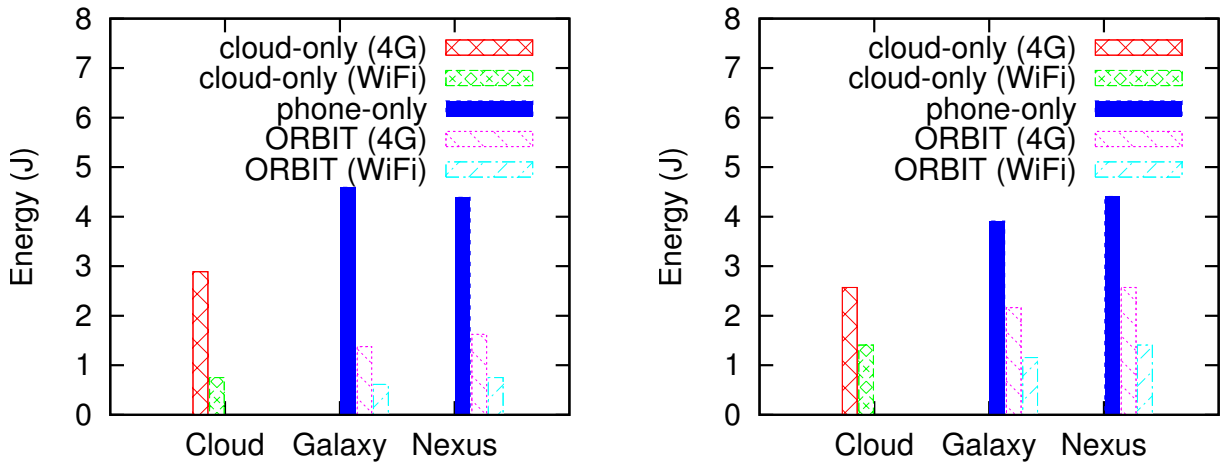


Figure 2.22: The block diagram of the Multi-camera 3D reconstruction application.

2.8.3 Multi-camera 3D reconstruction

The final case study is inspired by Phototourism [Snavely et al., 2006] and involves opportunistic sensing wherein smartphone-equipped robots capture location-based images to collaboratively reconstruct a 3D structure. Compared to previous two case studies, this application is partitioned cross over three tiers. The captured image is partially processed on the phone and the remainder of the processing as well as the distributed tasks are offloaded to the cloud server. Once an image has been processed, the robot is directed to move to a new spot to capture a new image. In addition to CPU, we also account for radio power consumption in this case study. Fig. 2.22 shows the task structure of this application. The cloud server is emulated by a Sun Ultra 20 workstation.

Effectiveness of Task Partitioner: For this case study, with the addition of the cloud tier and more complex input data to the sensing application, the communication delay between the smartphone and the cloud server and the complexity of input data impact the partitioning result. We evaluate the effectiveness of the task partitioning algorithm by comparing ORBIT with the phone-only and cloud-only baselines. In Fig. 2.23a and Fig. 2.23b two cases with different input images are compared: a) house image: a bigger image (in terms of number of pixels) with less complexity (in terms of number of SIFT features), and b) kermit image: a smaller image with higher complexity.



(a) House image (1280x722px)

(b) kermit image (640x480px)

Figure 2.23: The results of various partition schemes.

The difference between the partitioning assignments between these two cases is the assignment of the SIFT task. For the house image the results show that it is more energy efficient to run SIFT on the phone, because: 1) the image is less complex and thus SIFT runs faster and consequently causes the application to consume less energy, and 2) it would consume more energy to transmit the large image to the cloud for the SIFT processing. For the kermit image, it is more energy efficient to run SIFT in the cloud because it is a smaller image with more SIFT features. Thus, in both cases ORBIT comes up with the most energy efficient partition. In addition, this result demonstrates that ORBIT considers the execution time of data processing tasks not only as a function of input size but also as a function of input content. Existing task partitioning approaches [Newton et al., 2009, Cuervo et al., 2010a] often do not address the two affecting factors.

2.8.4 Discussion

These case studies demonstrate the generality of ORBIT's design. In particular, the three example applications differ significantly in the task structure, computation intensity of tasks, delay requirements, input data and the tiers involved in task partitioning. Overall, ORBIT can achieve energy saving of up to 50% compared to baseline approaches.

An interesting observation from case studies 1 and 2 is the system power consumption is highly probabilistic. For instance, whether the power-hungry image processing is needed in case study 1 depends on the decision of the first-stage IR/Sonar sensing which is subject to false alarms and misses. However, such runtime dynamics are unknown to ORBIT at the design time. As a result, ORBIT partitions the tasks according to the worst-case scenario in which a target is assumed to be present. A possible improvement would be to provide ORBIT runtime feedback such as the detection history and estimation of system detection performance. This would allow ORBIT to optimize the wiring of tasks and priorities of tasks to reduce power consumption. Such runtime adaptation is supported by ORBIT due to its flexible task partition and dispatch framework.

Case study 3 suggests the input data play important roles in the partitioning result. For example, when the input data is not too complex the tasks may not be offloaded to the cloud. However, the same image processing task may be offloaded to the cloud if it receives a complex input image. Making this decision at runtime, shows ORBITS intelligent flexibility.

2.9 Summary

This Chapter presented ORBIT, a smartphone-based platform for data-intensive, embedded sensing applications. ORBIT features a tiered architecture, in which a smartphone is optionally interfaced with an energy efficient peripheral board, and a cloud server. By fully exploiting the heterogeneity in the power/latency characteristics of multiple tiers, ORBIT minimizes the system energy consumption, subject to upper bounded processing delays. ORBIT also integrates a data processing library that supports high-level Java annotated application programming. The design of this library facilitates the resource management of the embedded applications and provides programming flexibility through adaptive delay/quality trade-off and multi-threaded data partitioning mechanisms. ORBIT is evaluated through several benchmarks and three case studies: seismic sensing, multi-camera 3D reconstruction and visual tracking using an ORBIT robot. This extensive evaluation demonstrates the generality of ORBIT's design. Moreover, our results show that ORBIT can save up to 50% energy consumption compared to baseline approaches.

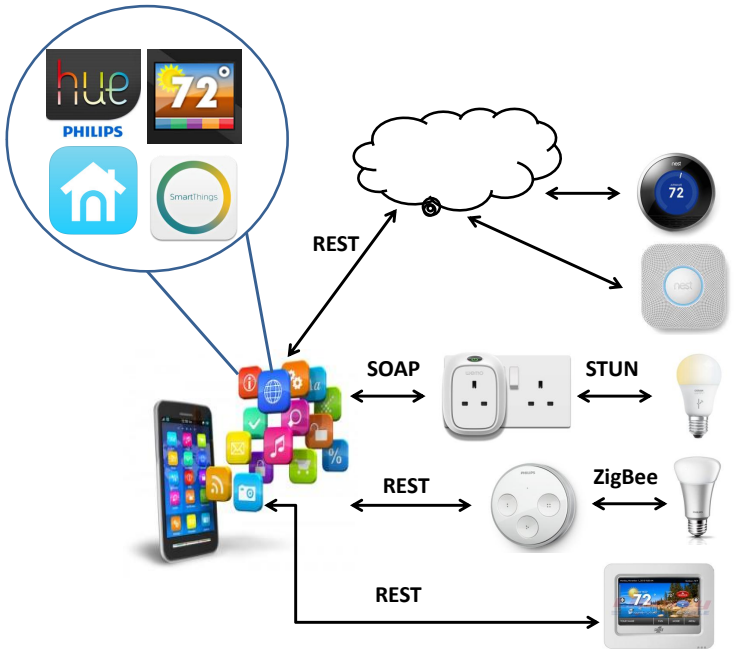
Chapter 3

SPOT: A Smartphone-Based Platform to Tackle Heterogeneity in Smart-Home Systems

3.1 Introduction

The vision of smart, connected homes has been around for decades. In this vision, users easily perform tasks involving diverse sets of devices in their home without the need for painstaking configuration and custom programming. For example, imagine a home with remotely controllable lights, air-conditioning systems, cameras, windows, and door locks. It should be easy to set up this home to automatically adjust windows and lights based on the outside temperature and lighting or to remotely view who is at the front door and then open the door. While modern homes have many network-capable devices, applications that coordinate them for cross-device tasks have yet to appear in any significant numbers.

Smart-homes differ in terms of appliances used there and how these appliances are connected and operated by users. The lack of global standards for smart appliances communication, control and data management results in highly fragmented systems consisting of proprietary solutions provided by each device vendor. Thus, users are required to use different control interfaces, e.g.,



(a) mobile apps (b) communication (c) appliances

Figure 3.1: Heterogeneity in today's smart-home systems. Each appliance in (c) requires its own app as shown in (a) that communicates with the appliance using its own protocol via cloud, a bridge and/or directly as shown in (b). Each appliance in (c) has different functionality and each smartphone app in (a) does not support appliances for more than one vendor nor share data with other apps. The user has to switch between apps to operate different appliances.

mobile apps, to interact with smart appliances in their homes or are forced to use devices sold by a single vendor. For instance, a user needs to separately operate Philips HUE [Phi,] app and WeMo [wem,] app to control lighting and smart power plug in the same room. Recently a cloud-based solution for multi-vendor IoT interaction, such as IFTTT [IFT,], has appeared, but user's flexibility is still limited to what the service provider offers. Moreover, third-party involvement might raise privacy concerns. Therefore, a user-centric platform that enables users to manage, monitor, and control heterogeneous smart appliances without relying on external parties is highly desired.

In this Chapter, we expand the scope of our study to smart-home systems as another emerging class of data-intensive sensing applications. We propose and demonstrate a platform built on a dynamic device driver abstraction model that tackles different aspects of heterogeneity in smart

appliances and smart-home systems. The device-control capability can be defined and expanded by means of "device driver" that are composed using XML or annotation-based JAVA APIs. The specification of the driver will be made public so that device vendors, open source community, or users themselves can create or modify the drivers. Besides communication with smart devices, our device driver framework implements automated generation of appropriate graphical user interface based on the driver definition of each device, which can minimize the developer's implementation efforts and improves the user experience by means of consistent look-and-feel throughout the system. Moreover, our framework provides a unified abstraction of the data structure in different smart devices that helps application developers to build "whole-home" applications that involve a diverse set of appliances.

In addition to providing a single point of control over multiple smart devices, our platform enables centralized collection of data about smart appliances used in the entire household, such as historical data of device statuses (e.g., on/off and set point) and control operations, as well as analytics and utilization of such data for energy efficiency, home automation, automated demand response, and so forth. We believe a generic platform like this is an important step toward a smart-home ecosystem from which smart appliance vendors, application developers, and users can benefit.

In summary, we makes the following contributions. First, we conduct a systematic study to understand the characteristics of smart-home appliances as well the opportunity and challenges for using smartphone as the central gateway to control smart-home appliances. The result of our study shows multiple aspects of heterogeneity in smart appliances. Second, we provide a flexible, extensive and extensible device driver model that supports a number of smart appliances available on the market. The driver model addresses multiple types of heterogeneity observed in our study. Third, we provide the design and implementation of the proposed platform as a smart-home system

that loads the drivers at runtime along with a dynamic user interface adaptive to the features of each appliance. Lastly, we demonstrate the generality and flexibility of our system by presenting our experience in prototyping the drivers for several real appliances as well as a cross-device home application. We also discuss examples of other home applications that we have prototyped on top of our platform.

3.2 Related Work

While we draw on many strands of existing work, we are unaware of a system similar to SPOT that provides a unified solution for heterogeneity problem that smart-home systems suffer. We categorize related work into the following groups:

Mobile apps for smart-home appliances: Many commercial off-the-shelf smart appliances are shipped with mobile apps that allows users to remotely control them, such as Nest Thermostat and Smoke detector [Nes,], Philips [Phi,], and GE Brillion [GE,]. However, these apps only support only their own devices. Such a vendor-centric solution requires users to use multiple apps. In contrast to these systems, SPOT provides a device/vendor-independent, user-centric system that dynamically supports multiple devices of different types (e.g., thermostats, lighting, home security), from different vendors.

Solutions based on hubs and central controllers: There are some solutions using hub devices for enabling centralized control and management of multiple appliances, such as Wink [win,] and SmartThings [sma,]. However, appliances still have to be designed and developed according to the proprietary specification provided by such solutions. Moreover, each hub-based system utilizes different communication protocols, making any interoperability highly challenging. Previous works also advocates using a central controller to simplify integration [Escoffier et al., 2008,Rosen

et al., 2004, Dixon et al., 2010]. These works have different scopes than SPOT. For instance, Rosen et al. [Rosen et al., 2004] focus on providing context such as user location to applications. Other systems have employed services in the home environment. iCrafter is a system for UI programmability [Ponnekanti et al., 2001]. ubiHome [Ha et al., 2007] aims to program ubiquitous computing devices inside the home using Web services. However, these systems only focus on the programmability of smart appliances and do not reduce the burden of users in terms of controlling diverse appliances in a home.

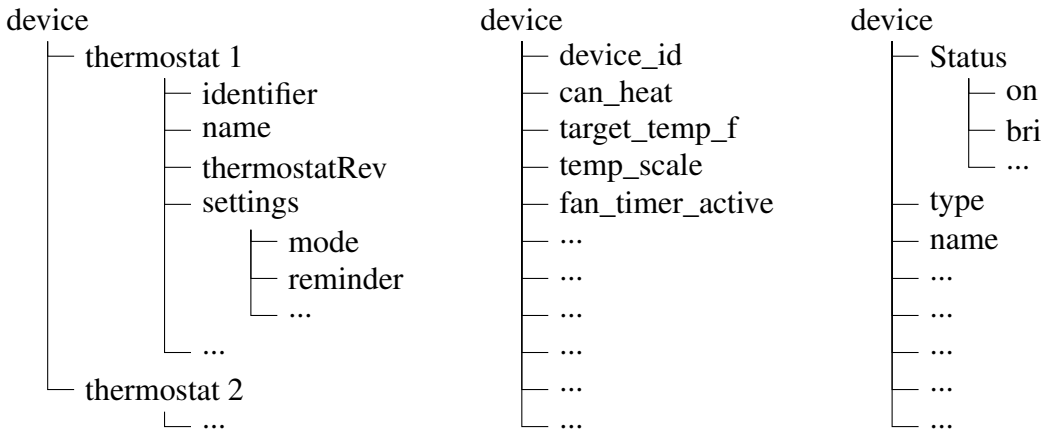
Multi-appliance platforms: Many commercial home automation and security systems integrate multiple devices in the homes. For instance, HomeOS [Dixon et al., 2010] provides a PC-like abstraction for digital devices to users and developers. But such system only supports devices with existing drivers and does not provide any dynamic driver loading mechanism. Control4 [Con,] only offers support for its own devices (and a limited set of ZigBee devices); a limited form of programming between devices. Furthermore, the technical complexity of installing and configuring Control4 and other systems alike (e.g., HomeSeer [Hom,], Elk M1 [ELK,]) can be handled only by professionals. EasyLiving [Krumm et al., 2000] is a monolithic system with a fixed set of applications geared for an specific domain (visual tracking via multiple cameras). Similarly, IFTTT.com offers the ability to define or download several predefined "recipes" in terms of "if this then that", allowing users to have access to shared configurations for devices. However, IFTTT.com only supports a certain number of devices, called channels, and does not scale based on user preferred devices unless new channels. In addition, reliance on a third party may cause privacy concerns given the sensitivity of data collected by the devices in the home. In contrast to such systems, we focus on building a device-independent system that can be extended easily with new devices and applications by non-experts.

3.3 Requirements and Challenges

Recent years, we can find a growing number of smart appliances on the market. However, these appliances exhibit significant heterogeneity in terms of communication protocols and architecture, device functionalities, programming abstraction and etc., owing to the lack of dominating standards in smart-home technologies. In this section, we will highlight different kinds of heterogeneity among smart appliances that brings challenges for different groups including *users*, *home application developers* and *appliance vendors*. We also discuss requirements to address each kind of heterogeneity based on our investigation of commodity smart appliances available on the market. We face the following major heterogeneities in building smart-homes:

Diverse appliance control apps: Currently each modern smart appliance comes with its own smartphone app. These smartphone apps do not provide similarity in user interface nor any interaction with other apps. Consequently users have to switch between multiple apps when they want to operate different devices which deteriorates the user experiences. Users need to learn specific apps functionalities in addition to appliance functionalities. Such diversity in appliance control apps presents barriers for adoption of smart appliances.

Communication interface: The communication interface for smart appliances varies a lot in protocol and architecture. Although most of smart appliances allow remote control via HTTP (or HTTPS) over WiFi, messages conveying control commands for each device differ. Many devices implement RESTful APIs with JSON messaging formats, while a few others, such as WeMo devices [wem,], only support SOAP messaging that exchanges XML messages. Another aspect of heterogeneity is communication architecture. As depicted by Fig.3.1, there are roughly three different communication architectures: appliances are accessible directly via local IP address in the home network; appliances require a dedicated "hub" to communicate with; and appliances are



(a) Ecobee thermostat (b) Nest appliances (c) Philips light

Figure 3.2: Heterogeneity in programming abstraction

controlled via vendors provided cloud service. For example, thermostats from Venstar [Ven,] and Radio Thermostat [Rad,] are controlled from devices in the same WiFi network using local IP address. Similarly, Wink [win,], SmartThings [sma,] Philips HUE light [Phi,] using local IP address but provide a gateway hub that implements WiFi interface for remote control. On the other hand, Nest's thermostat/smoke detector and Ecobee's thermostats [eco,] are controlled remotely (even from outside of a user's premise) via their own cloud services. Other aspects of heterogeneity observed in communication lie in security and user authentication mechanisms. Nest and Ecobee thermostats require users to use OAuth2 authorization token [Hammer-Lahav and Hardt,] while Philips HUE light simply relies on username/password only. We also found diversity in appliances initial setup. Many appliances require a simple configuration and use a static server port to accept control commands. Thus, the external system controlling such appliances need to be configured with the devices' IP addresses and port numbers, reconfiguration is rarely necessary unless IP address of devices changes. However, there are appliances like WeMo power plug [wem,] in which the port number is randomly selected among non-privileged ports and changes upon device restart etc. In this case the app provided by Belkin uses UPnP service discovery to identify the WeMo

device's IP address and port number. Besides WeMo devices, Philips HUE also adopts a device discovery service.

The heterogeneity in communication architecture and protocol presents serious hurdles for developers to develop genetic applications that suits multiple appliances from different vendors. From the developer's point of view it is highly desirable to standardize communication interface and messaging schema which are essential for building extensible applications. Moreover, lack of services like device discovery in majority of smart appliances requires users to perform the entire setup process for most of appliances they use, which is not necessarily a trivial process for non-expert users. This brings barriers in adoption of smart-home appliances and diminishes users ease-of-use.

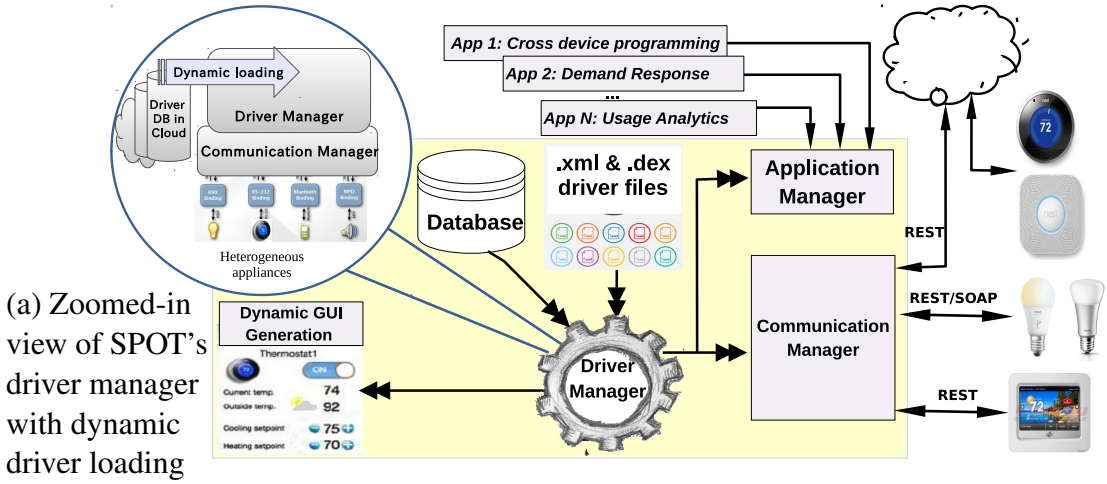
Programming abstraction: Even among devices using same protocol to communicate e.g., RESTful with JSON messages, variable names and the structure that forms the control command are not standardized. For example, to access set-point configuration, which is the most common variable in setting a thermostat target temperature, Nest thermostat uses a variable named *target_temperature_f* or *target_temperature_c* depending on what unit is used, while Radio thermostat uses either *t_heat* or *t_cool*, and Venstar thermostat uses either *heattemp* and *cooltemp* depending on the thermostat's operation mode. This means not only different appliances adopt different naming for the same setting, but also the structures of control messages are different. Fig. 3.2 and 3.3 illustrate different variable naming for different appliances. In addition, several appliances confine their settings by defining dependencies between the values of different variables. For instance, in Venstar thermostat all control messages with *mode* must include *heattemp* and *cooltemp* variables or when setting *mode* to $Auto$, *cooltemp* must be greater than *heattemp*, while in Nest thermostat the variables are set independently. Smart appliances also vary in terms of their internal data

```
Ecobee thermostat: "thermostat": {"settings":{"heatRangeHigh" : "72", "hvacMode" : "on"}}
Radio thermostat: {"tmode" : 1,"t_heat" : 72}
PhilipsHUE light: {"on" : true, "sat" : 255, "bri" : 255, "hue" : 1000}
Venstar thermostat: {'mode': 3, 'fan': 0, 'heattemp': 75, 'cooltemp': 78}
```

Figure 3.3: Examples of heterogeneity in message schema in setting different configurations

structure. Most commonly, the appliances utilize a hierarchical data structure for their variables. Fig. 3.2 illustrates the data structure of three common smart appliances i.e., Nest and Ecobee thermostats, and PhilipsHUE light. As depicted in the figure the grouping of variables as well the relationship in the hierarchy are different. From Fig. 3.2 and 3.3, we see depending on the data structure designed for the appliance the control messages are crafted differently. Fig. 3.3 depicts different messaging schema for different appliances. Unlike Radio thermostat and PhilipsHUE light that support a flat message structure, Ecobee thermostats adopts a "nested" structure. The difference in the data structure together with differences in the messaging schema and variables naming, makes it challenging for developers to design a common data management system for multiple appliances.

The above heterogeneities in control apps, communication interface and programming abstraction bring significant challenges in designing smart-home systems. These challenges motivate us to develop a platform that is universal to different appliances and yet flexible and extensible. Commercial off-the-shelf (COTS) smartphones offer several salient advantages that make them a promising system platform for smart-home applications. The advantages include rich computation and storage resources, multiple network interfaces and sensing modalities, highly increasing availability in each home and low cost. Moreover, smartphones come with advanced programming languages and friendly user interfaces, such as touch screen to enable rich and interactive display, unlike the limited user interfaces of existing smart-home hubs. Modern smartphones have



(a) Zoomed-in view of SPOT's driver manager with dynamic driver loading mechanism

(b) SPOT components on the smartphone (yellow area) (c) appliances

Figure 3.4: SPOT System Architecture

powerful multi-core processors which makes it possible to run analytical applications to provide better privacy guarantee to the users sine personal information does not need to be transmitted to the cloud. Another key advantage to use smartphones to control modern appliances is the fact that smartphones are personal devices and users are more comfortable in using it rather than learning to operate new appliances directly or to learn other interfaces e.g., hubs, to control appliances. Moreover, by providing a central unified smartphone app it is possible to log appliance usage data in one place and provide further analytical processing. It also enables the developers to develop whole-home applications that involves multiple appliances.

3.4 System Overview

In this section we present SPOT, which is designed to address the above major challenges. The key design principle of SPOT is to leverage the availability of smartphones in any house to support multiple heterogenous smart appliances and to facilitate the transition of current homes to smart-homes. Moreover, SPOT relies only on the out of the box functionalities of commercial-off-

the-shelf (COTS) smartphones, without requiring kernel level customization or rooting the device. This not only minimizes the burden on the application developers, but also ensures the compatibility with the diverse smartphone models. SPOT is designed as a mobile app that can be optionally backed by a cloud server as illustrated in Fig. 3.1 and 3.4. It enables the efficient collaboration between the smart appliances and smartphone to accomplish connected smart-homes, including smart appliances control, home energy management and automation, and centralized usage analytics. In the following, the major components of SPOT are described.

Appliance driver model and dynamic driver loading: SPOT provides a generic model to define the smart appliance properties and communication interfaces in a unified structure referred to as *driver*. A driver defines each variable and configuration setting as well as the detail of communication protocol and access control for each variable, the way the variables appear in the user-interface, and the range of accepted values for each variable. There are two types of appliance drivers: XML driver or Java driver using SPOT API. SPOT implements mechanisms to load the drivers dynamically at runtime. Fig. 3.4a depicts dynamic driver loading mechanism.

Dynamic user interface (GUI): In order to enable users to control smart appliances SPOT generates a specific GUI for each appliance based on specific appliance variables and configuration settings. The information about how the GUI is generated at runtime for each appliance and what variables are shown in GUI as well as the access control information are defined in the XML driver. By dynamically generating the GUI at runtime, SPOT enables access to any smart appliances without requiring to change the app.

Appliance discovery and status consistency: SPOT requires users to add smart appliances in the setting to be able to access to them. To facilitate this process, we implement a service discovery protocol based on UPnP that finds the smart appliances available in the home network and maps

the discovered appliances with the associated device driver. Moreover, SPOT adopts a monitoring scheme that polls the status of smart appliances periodically and keeps the usage information in an internal database. This enables application developers to implement data analytics about the appliances usage and extract information like daily pattern of user's appliance usage. SPOT can report this information e.g., on/off status, setpoint value of the thermostats to the cloud server for further data processing, if desired.

The SPOT design has several key advantages. First, it benefits *users* by providing one single gateway to their smart-home appliances and applications. It eases device setup and management and provides a central place for users to trace their activities e.g., track their energy usage. Moreover, users can benefit from accelerated device support as the open XML-based device driver framework is expected to encourage contribution from open-source developer community. In addition, the dynamic, adaptive user interface provided by SPOT improves the users experience and decreases their learning curve. Second, it benefits home appliance *vendors* by supporting the features and intelligence common in many appliances in one smartphone-based platform. Therefore, vendors do not need to provide all of these features in their appliances leading to simpler design and reduced product price. Vendors only need to provide a device driver according to our XML schema to enable the platform to leverage the device capabilities. Third, SPOT benefits *application developers* by enabling them to build applications that support multiple appliances from different vendors. SPOT also addresses each kind of heterogeneity outlined in Section 3.3 and brings a data abstraction and a unified data structure across heterogeneous appliances. This consequently reduces the burden on the application developer to deal with appliance specific interfaces and data structures. Lastly, as a gateway to the smart-homes that operates multiple diverse appliances, SPOT allows third party service providers i.e., utility companies, to access the user's data and provide services in a unified fashion.

3.5 Design and Implementation

The core component of SPOT is the appliance driver layer that abstracts the access to heterogeneous devices. In this section, we elaborate the design and implementation of such driver mechanism. SPOT supports two types of driver implementation, namely XML based and Java library based. However, the discussion primarily focuses on the former owing to the space limitation.

3.5.1 XML Driver Model

We designed a simple XML-based driver model for SPOT. This driver model defines multiple driver units, where each *unit* indicates an elementary appliance data unit referred to as *variable*, or an elementary appliance function, such as the details of protocol to communicate with the appliance to retrieve data or change a setting by sending a command. An appliance driver is composed of a number of driver units specifying the sequence of actions and list of variables based on the appliance's specific design and functionality. Such a driver model offers several benefits. First, by having a generic driver model, we can achieve extensible and universal mobile based appliance control without requiring the knowledge of the whole app or the source code of the app or the design detail of smart appliances. Second, the system can dynamically load the device driver into the app during runtime without the need to change the app structure or even downloading a new version of the app. More specifically, the XML drivers can be installed from the cloud via http or a smartphone's local storage and then parsed by the app according to the standardized XML schema. Third, by having the notion of driver units, we can build a unified system by addressing different heterogeneity related to different driver units. Fourth, the driver model can significantly simplify the application development and saves the efforts of users, especially for those who are not familiar with embedded system design and app programming. Users or application developers can thus

implement cross device applications using the same abstraction provided by the driver model without dealing with appliances specific programming APIs. In particular, SPOT presents application developers a single programming abstraction without burdening them with low-level details such as the data structure for each appliance or how each appliance's specific communication protocol works. The high-level view of the XML driver is shown in Fig. 3.5.

The key component of SPOT's driver model is multiple kinds of units. Specifically, a driver unit can be either a *data unit* or an *action unit*. The data units are classified into *common data* or *variables* and the action units are classified into three groups: *read actions*, *write actions* and *authentication actions*. To capture the fundamental role of each driver unit we explain each of them as following.

3.5.1.1 Driver Units

Variables: The *variables* driver unit is the most important unit in the SPOT driver model. This driver unit contains the list of variables and the detail information about each variable. A *variable* is referred to any elementary configuration setting or data element that can be read or set from/to a smart appliance. The XML schema snippet illustrated in Fig. 3.6 details the data structure in the driver model. Most of the elements are self-explanatory. *canonicalName* is enumeration, such as "Status", "SetPoint", "CurrentTemperature", etc. for thermostats, and tells the app the meaning of the variable. Such semantic mapping is needed to address heterogeneity in variable names and allows the app to process each variable in an appropriate way. *parent* is used for defining hierarchical relationship among variables to address the heterogeneity in the data structure demonstrated in Section 3.3. The *uiType, uiHelperText, uiCaption* are used in dynamic GUI creation.

Common data: The common data unit defines common data types about *all* kinds of appliances.

```
<?xml version="1.0" encoding="UTF-8" ?>
<drivers>
  <driver>
  <!-- common action unit-->
    <driverName>Philips HUE</driverName>
    <deviceType>Light</deviceType>
    <driverVendor>Philips</driverVendor>
      <!-- read action unit-->
    <read>
            ...
   </read>
      <!-- write action unit-->
    <write>
            ...
      </write>
      <!-- variables data unit-->
   <variablesList>
      <variable>
                ...
           </variable>
      <variable>
        ...
      </variable>
            ...
   </variablesList>
            ...
   </driver>
</drivers>
```

Figure 3.5: XML driver of the Philips HUE light

Currently this data unit is simple and includes the driver name, appliance type e.g., HVAC, lighting, and the appliance vendor information. The common data is used in SPOT to categorize the devices into its data structure for group management and data analytics. The XML schema for the common driver unit is simply implemented as:

```
<!-- variables data unit-->
<xs:element name="variablesList">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="variable"
       maxOccurs="unbounded" minOccurs="0">
         <xs:complexType>
           <xs:sequence>
             <xs:element
             type="xs:string" name="name" max="value" min="value"/>
             <xs:element
             type="xs:string" name="canonicalName"/>
             <xs:element
             type="xs:string" name="parent"/>
             <xs:element
             type="xs:string" name="type"/>
             <xs:element
             type="xs:boolean" name="writePermission"
             minOccurs="0"/>
             <xs:element
             type="xs:boolean" name="showOnUi"/>
             <xs:element
             type="xs:string" name="uiType"/>
             <xs:element
             type="xs:string" name="uiHelperText"/>
             <xs:element
             type="xs:string" name="uiCaption"/>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 3.6: The snippet of the XML schema for the SPOT's driver model

```
<!-- common data unit-->
<xs:element type="xs:string" name="driverName"/>
<xs:element type="xs:string" name="deviceType"/>
<xs:element type="xs:string" name="driverVendor"/>
...
```

Figure 3.7: The snippet of the *common* driver unit in the driver model

Read action: A read action unit specifies the detail of the communication protocol and its settings for reading the device data variables. As it is shown in the following XML schema snippet, this driver unit specifies the http method e.g., get, post that needs to be used to request the variable values from the appliances. It also specifies the base URL that the read request has to use and any extension to the base URL to create to access each variable. The read action unit also indicates the pattern of the read response, e.g., JSON, XML or MIME (*responsePattern*). Thus, the read action driver unit provide all the required information for SPOT to establish a success communication to each appliance and retrieve its data. *uriExtensionPattern* can be used to attach additional information as part of URL, e.g., query string in http GET requests.

```
<!-- read action unit-->
<xs:element name="read">
   <xs:complexType>
     <xs:sequence>
       <xs:element type="xs:string" name="httpMethod"/>
       <xs:element type="xs:anyURI" name="baseUri"/>
       <xs:element type="xs:string"
                name="uriExtentionPattern"/>
       <xs:element type="xs:string" name="responsePattern"/>
     </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 3.8: The snippet of the *read action* unit in driver model

Figure 3.9: The *write actions* using driver model

Write action: The write action unit in the driver describes the communication method to set or change the current value of any variable (with the write permission) in each appliance. It indicates whether the new value and the variable name have to be set in the URL extension or in the body of the http message. It also describes the pattern of the body of the message (*bodyPattern*) if the write command has to be defined in the body part of the message. Currently the SPOT driver model support several body patterns including json, xml, mime and url-encoding. The snippet of *write action* unit is shown in Fig. 3.10

Authentication action: As discussed in section 3.3, one of the aspects of the heterogeneity of smart appliances is their authentication and user management mechanisms. Therefore, it is clear that one of the essential parts of the driver model needs to define the authentication mechanism

```
<!-- write action unit-->
 <xs:element name="write">
   <xs:complexType>
     <xs:sequence>
       <xs:element type="xs:string" name="httpMethod"/>
       <xs:element type="xs:anyURI" name="baseUri"/>
       <xs:element type="xs:string" name="bodyType"/>
       <xs:element type="xs:string" name="bodyPattern"/>
       <xs:element name="httpHeaderFields">
         <xs:complexType>
           <xs:sequence>
             <xs:element type="xs:string" name="headerField"/>
       </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 3.10: The snippet of the *write action* unit in driver model

for each appliance. Our extensive exploration in several smart appliances suggest there are two major authentication mechanisms are adapted by the vendors namely as OAuth2 e.g., Nest and WeMo and simple username/password authentication e.g., Philips HUE light. SPOT supports both authentication mechanisms. At the high level, the former can be addressed by defining optional *auth* element, which stores token expiration time, URL for token refreshment, etc., while the latter can be done by including username/password as part of *baseUri* or *uriExtentionPattern*.

3.5.1.2 Device Driver Usage

Reading values from the appliance: SPOT can read/fetch variable values from the appliances via a network connection, using RESTful/SOAP protocols. SPOT initiates an HTTP request to an appliance, querying for list of variables listed in the XML driver, possibly using additional headers, using HTTP GET/POST methods as indicated, and possibly with a body as defined in the driver.

Upon receiving the HTTP response from the appliance, the payload of the response message is pattern-matched to the format indicated in the XML driver e.g., different JSON or XML formats. After pattern matcher verifies the message is well-formed the driver manager extracts the variable values and if some variables are indicated to be shown in the UI, it passes them to the user-interface component. The user-interface then presents the variables and their values based on the defined format using the indicated *UI component* e.g., as a text or as a value of the progressbar. Finally, if necessary, variables are stored in the SPOT internal database or processed based on *canonicalName* definition.

Setting/Updating the values on the device: In order to set values on the appliance (update the appliance configuration), SPOT sends the HTTP request to one or multiple fields possibly using additional headers, using POST/PUT as indicated, and possibly with a body message. If the new values for the variables are set by user from the user-interface, similar to the *read action*, SPOT maps the variable values to the accepted format and range defined for each variable in the XML driver. It then creates the message according to the settings defined in the *write action unit* and sends to the associated appliance. If necessary, the value of the variable is updated in the database upon receiving a ack from the target appliance. Also, the value of the variable is updated in the UI after a successful write action. Fig. 3.9 illustrates this process.

Dynamic UI Creation: A salient feature of SPOT is the Dynamic UI creation. In section 3.5.1 we describe how device drivers are abstracted and how one can easily craft an XML driver for a new appliance using our defined XML structure. The system then utilizes the XML-formatted driver and extracts the fields, access controls, and message formats to communicate with the device. Similarly, upon receiving the XML-formatted driver, the system reads the appropriate XML tags indicating what type of GUI components has to be generated and appear in the associated section

```
/* Annotation @UIElement specifies each UI
    component */
/** import mHEMS API **/
   @UIElement( uiType = UIType.Texview, helperText
        = "The Appliance Name" , caption=" Driver
        Name")
   private final static String _driverName =
        DLCDeviceDriverConstants.
        _DRIVER_VENDOR_PHILIPS+
        DLCDeviceDriverConstants._DRIVER_TYPE_LIGHT
         ;

   @UIElement( uiType = UIType.SeekBar, max=100,
        min=0, helperText = "Set the Brightness" ,
        caption=" Brightness " )
   private int _brightness;

   @UIElement( uiType = UIType.SeekBar, max=50, min
        =10, helperText = "Set the Saturation" ,
        caption=" Saturation ")
   private int _saturation;

   @UIElement(uiType = UIType.SeekBar, helperText="
        Set the Color", caption = "Light Color")
   private String _color;

   @UIElement(uiType = UIType.TgleButton, caption="
        On/Off" )
   private boolean _onOffState;
```

(a) JAVA specification

```
<variablesList>
 <variable>
      <name>ip</name>
      ...
      <showOnUi>true</showOnUi>
      <uiType>EditText</uiType>
      <helperText>
            e.g. 192.168.19.10
      </helperText>
      <caption>IP</caption>
 </variable>
 <variable>
      <name>on</name>
      ...
      <showOnUi>true</showOnUi>
      <uiType>TgleButton</uiType>
      <helperText>
            The On/Off State
      </helperText>
      <caption>On/Off</caption>
 </variable>
      ...
</variablesList>
```

(b) XML specification

Figure 3.11: JAVA and XML specifications for dynamic GUI generation

in the mobile app for any device that user wants to control. This information are obtained from the user-interface (UI) related tags, *uiType, uiHelperTex, uiCaption*, defined in the variable driver unit. The GUI components in the Android are all sub-classes of a parent class called View e.g., *Button, TextView, ProgressBar*, and by intelligently implementing the system the user-interface renders at runtime. This feature not only enables the users to control different variables from different appliances without requiring to design a separate app, but also provides the appliance vendors the ability to indicate what variables and how they want users to have access to (access control). By changing the values in UI-related fields in the XML driver, the GUI changes accordingly in the next run of the app without any need to change in the design of the app.

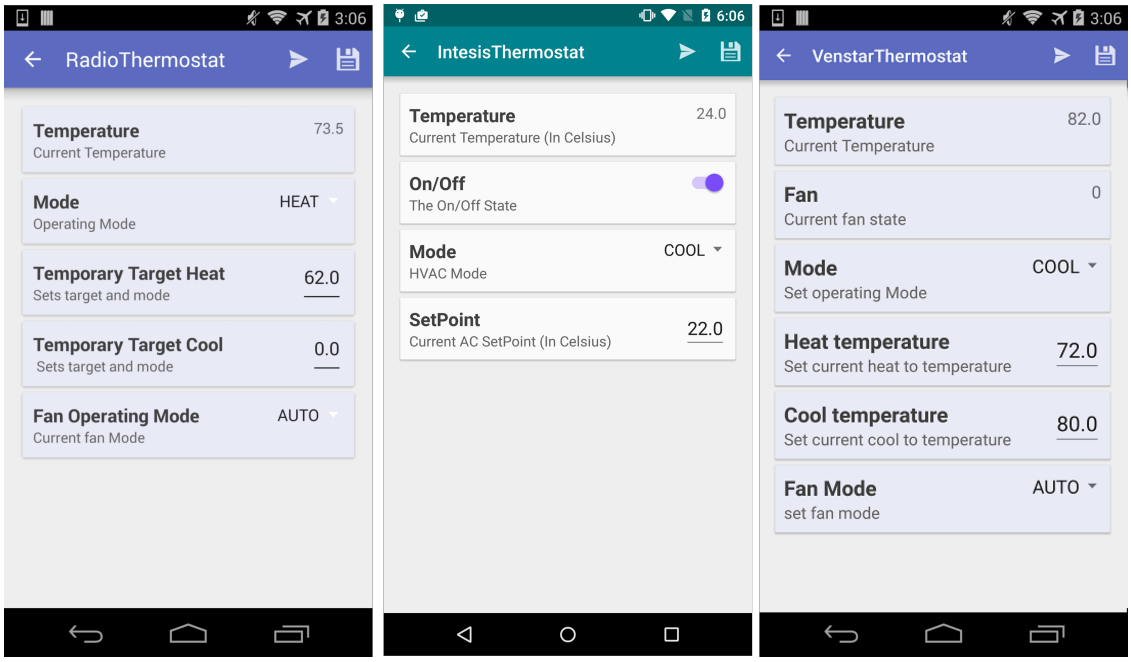


Figure 3.12: An example dynamically generated GUI in SPOT

3.5.2 Appliance Driver by SPOT JAVA Library

Besides the XML-based device driver framework, SPOT provides the application developer an API, using Java annotations. By using this API a driver developer implements the device driver as a Java class specifying the device variable as a field in the class and uses SPOT-provided annotations to annotate each variable. By annotations the developer indicates what information the system can fetch from the device and how they are structured e.g., RESTful URI. Moreover, by using the annotations, the developer can set what variables appear in the UI, what access level user can have e.g., read-only, read/write, and what kind of UI component should be used by SPOT for each variable. Also, more meta data like the default, maximum, minimum values for each variable can be specified. In addition, the driver can indicate the variables that have to be kept persistent in the database using the appropriate annotation tag. More importantly, the driver specifies the URI to access in read and write scenarios as well as the message patterns e.g., JSON/XML format, for

send and response messages to/from the device. Using SPOT annotation APIs, writing such a class is very simple. For example, only the variables and meta data for variables need to be defined, and no method has to be implemented. For instance, a snippet of a Java class generating a driver for the Nest Thermostat can simply be implemented as shown in Fig. 3.11a.

The java classes then must be packed as jar files and after submitting to SPOT, the system verifies the jar file and then creates a DEX module [DEX,] from the legitimate jar files. The DEX module is dynamically loaded to the system and become functional right after. The SPOT API provides a compact and efficient way to implement the appliance drivers, however it requires the programming knowledge and learning the API, and thus is a suitable method for the developers rather than the users, while this is not the case for the XML-based method.

3.5.3 Appliance/State Consistency

To effectively manage smart-homes, having up-to-date status of each smart appliance is mandatory. However, since the state of the smart devices can change via other mobile apps e.g., vendor specific apps or physically by changing their settings e.g., turning off the light via a wall switch, it's highly possible the devices states become inconsistent with the stored state in the SPOT database. In order to address this issue SPOT periodically synchronizes its database by polling the devices, using the mechanisms described in Section 3.5.1.2. However, these changes may not happen frequently and the state of different devices may not change with same frequency (e.g., one may change the setpoint of a thermostat more often than turning on/off a light in the bedroom). Thus in order to have a more efficient device/state persistency and to prevent draining the smartphones battery by unnecessary polls, SPOT can have an adaptive polling mechanism. Intelligent adaptive polling mechanism is left for our future work.

3.5.4 Appliance discovery and bootstrap

To minimize users' efforts to register new smart appliances on the system, SPOT supports device discovery mechanism using UPnP. By using cybergarage-upnp library [upn,], SPOT listens SSDP NOTIFY messages broadcasted by UPnP-capable smart appliances in the network. Through this message, the SPOT identifies IP address and port number of the appliance to which read/write commands are to be sent. Moreover, when the default device name is included in the message, the app automatically finds an appropriate device driver. If it is not found, the app can download it from the cloud. Further utilization of UPnP is part of our future work.

3.5.5 Application Manager

SPOT enables home-wide application execution by Application Manager component. Developers or hobbyists can use SPOT API to write home-wide applications and submit to the SPOT. SPOT checks the legitimacy of each application and dynamically loads them into the system (with the same technique to dynamically load annotation-based JAVA drivers). SPOT persists each application into the database and pass its handler to the Application Manager component. Application Manager is implemented as a background process and periodically checks for the active applications in the system, if the execution requirement of each application is satisfied then it executes the application. Since home applications involve one or more smart appliances and each appliance can be associated with one or more application, the application manager service ensures the persistency of device states and their configuration changes with the requirements of associated applications. This is performed by the interaction with device management and underline database management services.

3.6 Application Scenarios

To demonstrate the extensibility and universality of SPOT driver model and data abstraction, and the generality and flexibility of SPOT as a platform, we have prototyped three different smart-home applications. Each application demonstrates the advantages of having SPOT as a central platform for home automation systems and the opportunities it provides for application developers. Our goal is to demonstrate the capabilities and effectiveness of the platform and show the opportunities provided by SPOT rather than present novel applications.

3.6.1 Application 1: Cross-Device Programming

Unlike other contexts (e.g., enterprise or ISP networks), the intended administrators for the home networks are non-expert users.The management challenge is particularly noteworthy when it comes to inter-device control and applications that involve the entire smart-home or part of it but requires more than one device. SPOT provide a platform for the cross-device applications. For this purpose SPOT provides an application layer to enable the users and developers to design custom applications that operate home-wide. SPOT supplies application management service to execute the defined applications along with "trigger-target" application. The idea of this application is inspired by the IFTTT system [IFT,]."Trigger-target" application is defined by "trigger-target" rules. In trigger-target programming, end users specify the behavior of a system as an event (trigger) and corresponding action (target) to be taken whenever that trigger occurs. Both the trigger and the target can contain parameters that customize their behavior.

Each trigger and its target involve one device (trigger device and target device). Trigger also defines the circumstances that lead to its occurrence. For that a user selects a variable from the trigger appliance and the condition and a triggering threshold when he defines the trigger. User

also selects the target device, chooses one of its variables as target variable along with the target value for that variable when it defines the target. Once the trigger and target are defined a trigger-target rule is programmed and it becomes active immediately. Application Manager of SPOT then takes the rule and places it in the application pools to execute when the trigger occurs. For example, if target temperature set in Nest thermostat is equal to 70 then turn off the Philips HUE light. Without using SPOT, users can only program simple inter-device applications using third party cloud-based systems like IFTTT. However, involvement of third party systems might rise some privacy concerns for the users. Moreover, systems like IFTTT only provide interfaces to a limited set of appliances that are integrated with the implemented interfaces.

3.6.2 Application 2: Residential Automated Demand Response

Demand Response (DR) is a technology to enable balancing of electricity supply and demand. In DR, utility companies send signals to electricity customers to request their electricity demand curtailment, in order to handle the situation where significant demand peak is expected or when electricity price is high. By participating in DR, customers can gain monetary incentive or incur lower electricity cost. SPOT can benefit users (i.e., electricity customers) by facilitating DR participation by means of home automation. Specifically, SPOT works as a home gateway, which interacts with electricity utility company's DR server and controls smart appliances in a household according to the signal from the server. For instance, in the case of DLC (direct load control), which is probably the most popular DR program, SPOT can automatically change the operation state of the targeted appliance (e.g., turning off lighting and /or changing set point of an air conditioning system). Dynamic pricing, which encourages voluntary demand curtailment by adjusting electricity price during peak hours, also has gained popularity. In this case, SPOT interprets pricing information from a DR server and automatically reduces or shifts appliance usage. While tradition-

ally DR participation required manual control of individual appliances, which deterred penetration into residential sectors, SPOT is expected to significantly reduce the user's efforts and hence lower the barrier of participation.

We have implemented a prototype DLC app that implements OpenADR 2.0b protocol [Ope,], the globally-recognized standard for automated demand response, as well as SPOT for automated appliance control. Functionality to report appliance status (e.g., on/off and setpoint) is also implemented using SPOT so that the DR server can use such data to validate DR participation. Moreover, although all of the smart devices discussed in this work (listed in Table 3.1) are not OpenADR-compliant as of April, 2015, SPOT immediately allows integration of them into automated demand response systems.

3.6.3 Application 3: Central Usage Analytics

As explained in Section 3.5.1, SPOT provides a way for collecting status of all smart appliances in a household. In other words, a smartphone app can access the rich historical data regarding when each appliance was active. Such centralized data collection enables a variety of data analytics that is useful for users. Combined with the user's energy usage data, which can be obtained via services like Green Button Download My Data or Connect My Data [Gre,], the app can accurately label the energy consumption trace with activeness of operation status of each appliance, without requiring users' effort. Such labeled traces can be used, for example, for electricity load disaggregation, which is takes a whole-home energy signal and separates it into component appliances, and provides the user with fine-grained understanding of her own energy usage. Namely, the app can tell how much electricity is consumed by each appliance and how much electricity cost is attributed to it. Furthermore, other types of supervised data analytics can also be made possible.

Table 3.1: Smart appliances tested with SPOT

Appliance Name	Vendor	Type	Protocol	Architecture	Upnp	Auth.	Msg. schema	driver
Philips HUE light [Phi,]	Philips	light	RESTful	HUB	✓	user/pass	JSON	all[a]
Nest smart thermostat [Nes,]	Nest	thermostat	RESTful	Cloud	x	OAuth2.0	JSON	all
WeMo Switch [wem,]	WeMo	plug	SOAP	Local IP Add.	✓	x	XML	XML
Venstar thermostat [Ven,]	Venstar	thermostat	RESTful	Local IP Add.	x	x	JSON	all
UFO Smart Plug [ufo,]	UFO	plug	RESTful	Local IP Add.	x	x	JSON	all
Radio Thermostat [Rad,]	Radio Therm.	thermostat	RESTful	Local IP Add.	x	x	JSON	XML
Ecobee Thermostat [eco,]	Ecobee	thermostat	RESTful	Local IP Add.	x	OAuth2.0	JSON	XML

[a]: *all* means all three types of drivers (XML, Pure Java, SPOT annotation-based Java API) are implemented for this appliance.

3.7 Evaluation

To demonstrate the usability of SPOT as a platform, we conducted extensive analysis to evaluate SPOT performance in detail. Our goal is to achieve the latency that is low enough to handle application scenarios discussed in Section 3.6 and to offer scalability and throughput that can handle large, complex smart-homes. In this section, we report our system evaluation in different criteria from memory and CPU usage of SPOT as a smartphone app and the latency of appliance control command invocation to the overhead introduced by dynamic driver loading, appliance status polling, and so forth. We also look at the scalability of system to managed multiple smart appliances in smart-homes.

In order to show the universality of our driver model and SPOT app, we prototyped device drivers for multiple appliances on the market from major home appliance vendors. Fig. 3.1 show the appliances that are currently tested using our driver model and the app. As elaborated in Section 3.3, appliances vary in several different aspects including communication protocol and ar-

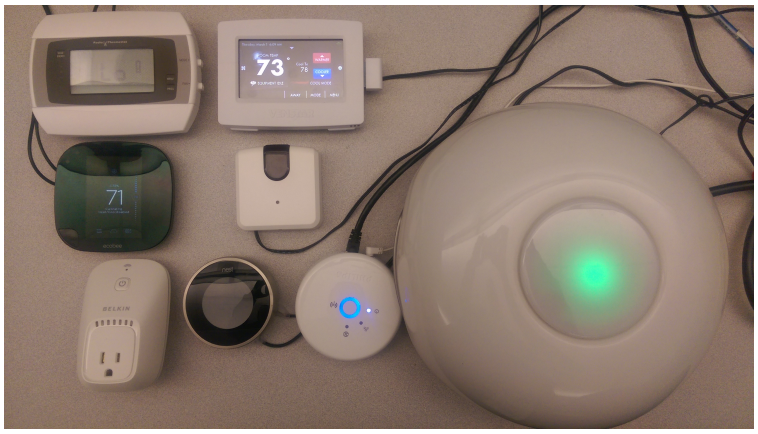


Figure 3.13: Smart appliances tested with SPOT

chitecture, messaging schema and data structure, authentication mechanism, and support of device discovery.

Comparison of XML-based and JAVA-based drivers: As discussed in Section 3.5, there are basically three ways of implementing an appliance driver: a) conventional driver, which is to implement each driver without using SPOT driver model e.g., a proprietary Java class for each appliance and implement all the functionalities and variables independently, b) using SPOT Java API (cf. Section 3.5.2) and c) XML-based driver (cf. Section 3.5.1). We compare the ease of programming a driver in terms of line-of-code(LOC) for each kind of driver for multiple appliances. Fig. 3.14 shows that the implementation code of the drivers using SPOT driver model (XML-based and API-based) is significantly shorter than the conventional driver. This benefit is regardless of the other benefits of using SPOT driver model such as dynamic UI generation and automatic DB connection. In addition, we note that implementing drivers using SPOT API is even shorter than using XML models. This is because in the XML model, as illustrated in Section 3.5.1, in order to define the driver units e.g., variables and action units, it requires to provide a few lines of XML code for each. However, using SPOT API one can define all the meta information about each driver unit in a compact line using SPOT API annotations (cf. Fig.3.11a). Moreover, the Java

drivers implemented by SPOT API do not require to implement the action units as opposed to the conventional driver. This is because the driver manager in SPOT provides actions e.g., read, write, authentications for each driver according to the driver model. It is important to note that although SPOT API provides a compact and efficient way to implement the appliance drivers, it is a suitable method only for the application developers rather than the users since it requires the programming knowledge and learning the SPOT API, while this is not the case for the XML-based method.

CPU and memory footprint on smartphone: We measure the CPU and memory footprints of SPOT. We select different features that SPOT provides. We use the utility application, System Monitor, to measure the CPU and memory usages. The CPU usage of SPOT is almost 0.0%. The memory usage is about 49 MB during silence, but reaching 108 MB when visualizing the reports. This is due to the dynamic increase of the heap space when data being loaded for visualization. The total size of the SPOT binary is only 9.69 MB.

Latency of appliance control command invocation: We also measure the latency of an operation invocation with different number of variables to set under two different situations: $a)$ when the smartphone and a smart appliance communicate directly on the local WiFi, e.g., Philips HUE light and Venstar thermostat, or $b)$ when the smartphone and the smart device communicate via the smart device's (or a third party) cloud e.g., Nest Thermostat. Our measurements show it takes less than 150 ms when the command is invoked over local WiFi and and it takes around 395 ms when it is invoked via Internet (device cloud). The low latency makes SPOT a suitable platform for applications with specific timing requirements. For example there are demand response services, (cf. in Section 3.6.2), such as Fast DR [fas,] that require the operations to the targeted appliances to finish within 4 seconds. Our results show that the command invocation time on SPOT is short enough to support this use case.

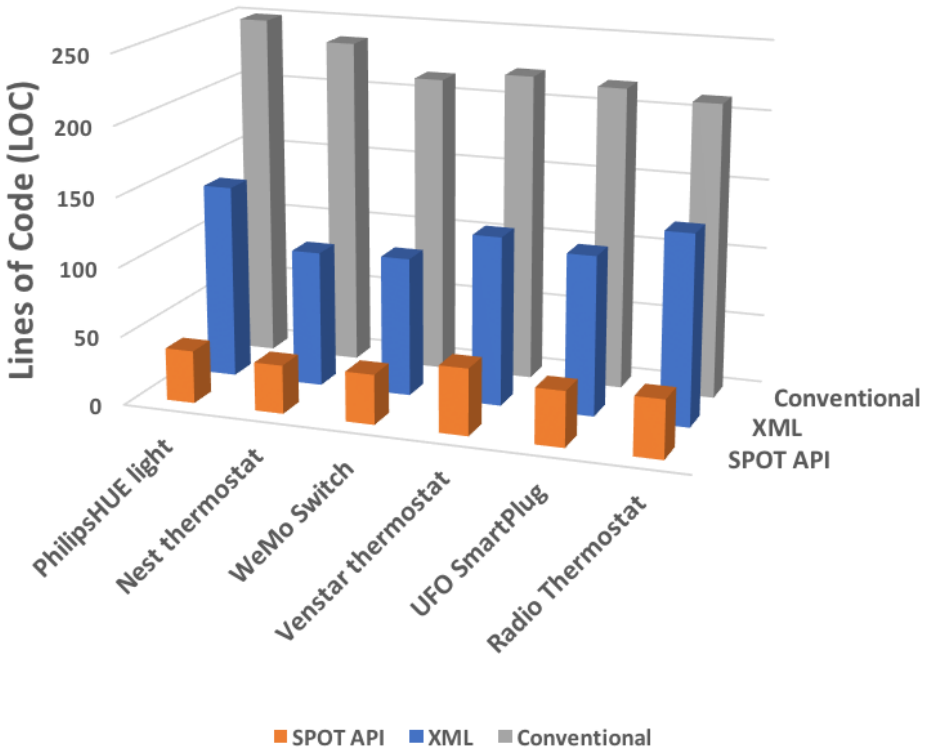


Figure 3.14: The length comparison (LOC) of different kind of drivers

Latency of dynamically loading drivers: Although loading and processing the device drivers into the system does not happen frequently, it must not interfere with user's experience as well as time-critical operations. We measure the time it takes to load and process the XML driver with 10 different variables. Our measurements on Galaxy Nexus, Nexus 5, Nexus 7 show that the drivers are loaded less than 25 ms. SPOT processes the XML drivers in a background thread and thus it does not interrupt the user's experience. Fig. 3.15 shows the latency of dynamically loading an XML driver versus the number of fields defined for the smart appliance.

In Section 3.5.2 we discussed that an alternative to XML-based drivers is to draft the appliance drivers as minimal Java class using the annotation-based SPOT API. These driver classes are packaged as DEX files and are loaded dynamically. We measure the time it takes SPOT driver manager to load the DEX drivers and process the annotations to extract the driver specific infor-

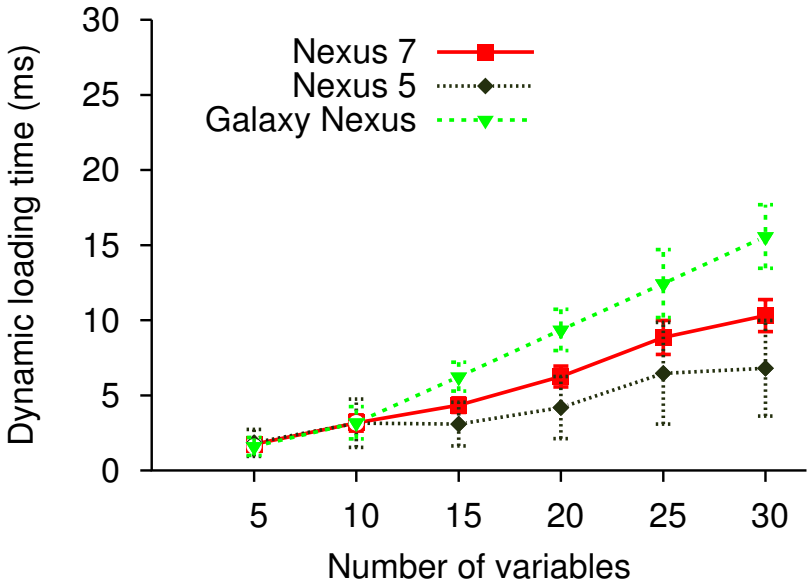


Figure 3.15: Latency of dynamically loading the XML drivers (error bars: standard deviation)

mation. Similar to Fig. 3.15, our measurement is done for drivers with 10 different variables on different smartphones. Our measurements shows the DEX drivers are loaded and processed in less than 20 ms. This process is an I/O process and it is executed in the background to prevent any user interruption or any interruption with the main thread.

DB query delay when fetching smart appliance state: In Section 3.5.3 we explained how SPOT stores the device states in a lightweight DB to provide appliance state consistency. We measure how long it takes to persist the appliance information and to retrieve its state from the database. Fig. 3.16 shows the latency of database query on various smartphones versus the number of appliance records in the database. We can see querying the DB when there are a few records of smart devices are stored takes less than 4 ms. Although this delay is very short, it can vary based on the number of records. To achieve a design that is scalable in terms of number of appliances and to have an smooth transition between different components of the system including the user-interface components, this I/O task, similar to other I/O and network tasks, is done in a background thread.

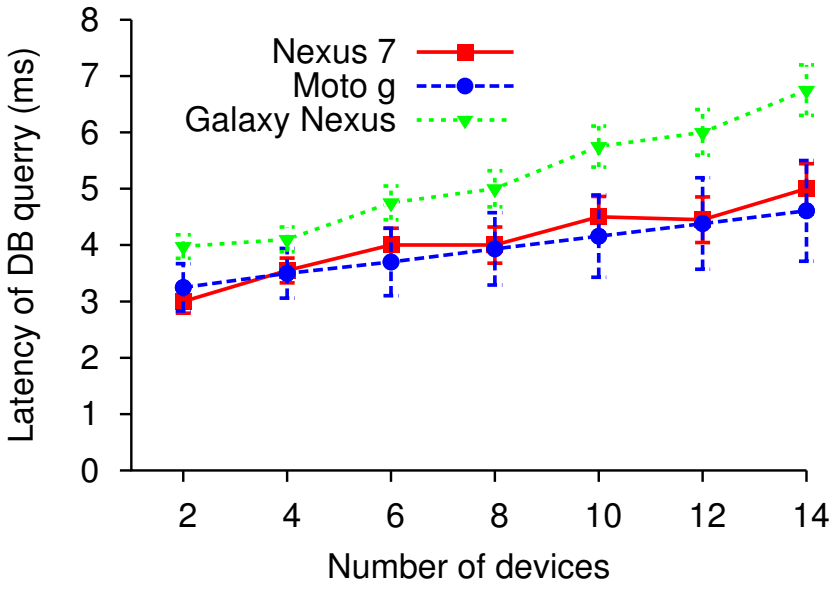


Figure 3.16: Latency of database query

The effect of polling appliances state: In addition to the latency of the DB query, another important factor related to the appliance state consistency is the frequency for polling the appliances. Polling too often could drain the smartphones battery quickly. In this section, we measure how polling affects the battery level when SPOT runs on Nexus 6. Fig. 3.17 shows the trend of battery level drop for different number of appliances and different polling intervals. For experiments with 3 devices, we used Radio Thermostat, UFO Smart Plug, and Philips HUE. For 6-device experiments, we additionally used Ecobee, WeMo Switch, and Venstar. Note that the separation between the lines for different experiment is made for better readability, and is not related to battery level differences. Intuitively it is expected the longer polling interval leads to a slower energy consumption while polling more appliances cause a faster energy consumption. As can be seen, even in the most aggressive case (polling 6 appliances every 1 minute), the battery level drops at most only about 1 percent per hour. This result suggests that frequency of this level is practical. Moreover, we can see by choosing a longer polling interval the rate of the smartphone's energy consumption decreases that agrees with our hypothesis. In addition we see when SPOT polls 6 appliances rather

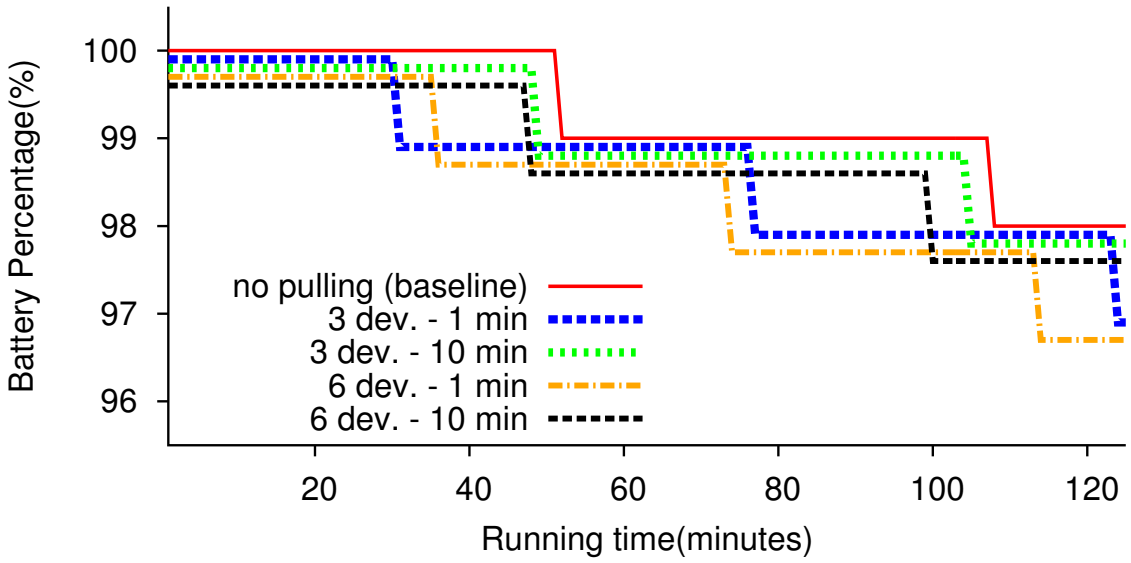
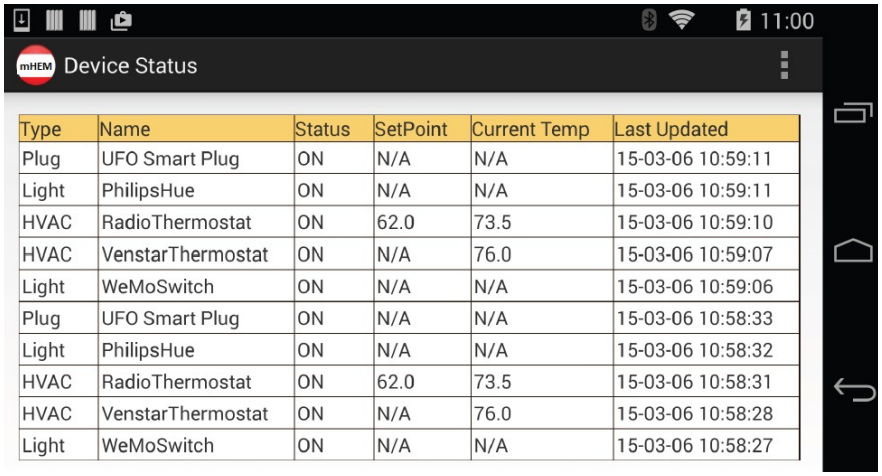


Figure 3.17: The effect of polling interval and number of appliances (devices) on smartphone's energy consumption. Shorter polling interval and more number of devices lead to higher rate in energy consumption of smartphone.

than 3 the drop rate in battery level increases as expected, but the increase in energy consumption rate is moderate (a few minutes). These results suggest, in this context, the polling interval plays a more important role than the number of appliances. Based on our experiments using 5 real devices (ufo, venstar, wemo, radio, hue), we confirmed that SPOT can handle over 800 device state pollings per minute. Therefore not only the polling mechanism is practical but also it is scalable to more appliances.

Fig. 3.18 is the screenshot of the page in SPOT that shows a part of appliance status stored in the SPOT internal DB. We can see multiple appliances and their defined type as well as their current status including on/off state, current setpoint, current temp (if any). By providing this central and consistent information about the smart appliances at home, SPOT facilitates the development of different kinds of whole-home applications. For instance, maintaining the consistent information about the appliances status not only is useful for home automation and is essential for users to control appliances, but also is crucial for applications like data analytics and demand response.



Type	Name	Status	SetPoint	Current Temp	Last Updated
Plug	UFO Smart Plug	ON	N/A	N/A	15-03-06 10:59:11
Light	PhilipsHue	ON	N/A	N/A	15-03-06 10:59:11
HVAC	RadioThermostat	ON	62.0	73.5	15-03-06 10:59:10
HVAC	VenstarThermostat	ON	N/A	76.0	15-03-06 10:59:07
Light	WeMoSwitch	ON	N/A	N/A	15-03-06 10:59:06
Plug	UFO Smart Plug	ON	N/A	N/A	15-03-06 10:58:33
Light	PhilipsHue	ON	N/A	N/A	15-03-06 10:58:32
HVAC	RadioThermostat	ON	62.0	73.5	15-03-06 10:58:31
HVAC	VenstarThermostat	ON	N/A	76.0	15-03-06 10:58:28
Light	WeMoSwitch	ON	N/A	N/A	15-03-06 10:58:27

Figure 3.18: SPOT records the state of appliances and maintains appliance/state consistent in its internal DB with frequent polling.

Latency of dynamically generating GUI: A salient feature of SPOT is its capability to dynamically create the appropriate graphical user interface (GUI) for controlling smart appliances. The components used in the GUI are indicated by designated tags in the XML driver or appointed Java annotation in the JAVA-API drivers. The components are specified for each field in the driver and has to be selected from the supported components available in the Android library. Moreover, different appliances can have different number of fields accessible in the UI. As a result the UI load time can vary for different number of UI components. Our measurements show the user interface load time is around 5ms and changes at most by 1ms when the number of components in the UI are up seven components. It should be noted that maintaining small variation in loading the user interface is essential for achieving good user experience.

The smoothness of displaying dynamic GUI: As an mobile app that provides a dynamic user interface for the users, drawing screen frames with a regular rhythm is essential for good performance and use experience. We analyze this by using an Android system tools, Systrace [sys,], which is particularly useful in analyzing application display slowness or pauses in rendering the

UI components. Typically the analysis of display components (UI threads) by Systrace is reported under *SurfaceFlinger* process as shown in Fig. 3.19. Having a regular rhythm for display ensures that UI components are smoothly appearing on the screen [sys,]. Fig. 3.19 illustrates the execution pattern of display component in SPOT. The regularity of *SurfaceFlinger* process suggests a smooth GUI rendering in the app. Moreover, the regular pattern in the CPU state in the upper panel of Fig.3.19 indicates that there is no other threads in the app, e.g., network communication, disk operation for DB access or loading the UI components like images, which may interrupt the rendering of the user-interface. This validates the efficient architecture of SPOT that achieves the smoothness in displaying the UI.

Latency of cross-device application runtime: A salient feature of SPOT is that it enables developers to create whole-home applications that involve and connect multiple smart appliances. In order to show the performance of such applications, we use the Application 1 discussed in Section 3.6.1. Fig. 3.20 shows the running time of this application for different number of smart appliances used. Note that this execution time only presents the application runtime delay and does not include the delay of command invocation to the appliances as the appliances response time can significantly vary owing to the diversity in their communication architecture. In this figure we can see having an smart-home application running on top of SPOT (cf. Section 3.6) will add at most 13 ms to the execution time when handling the information of 14 appliances. This shows the practicality of build whole-home applications.

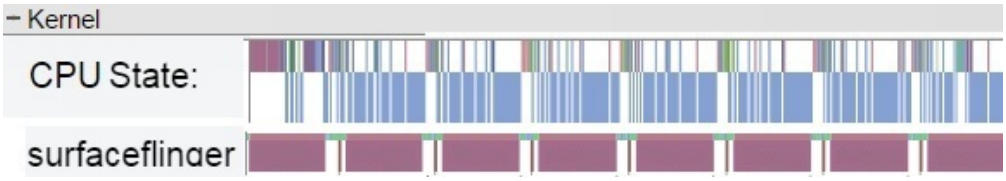


Figure 3.19: The smoothness of displaying GUI: The regular rhythm in *SurfaceFlinger* process indicates the smooth display rendering. The regular rhythm in the CPU state in the same period of time indicates no interference between the threads in the app.

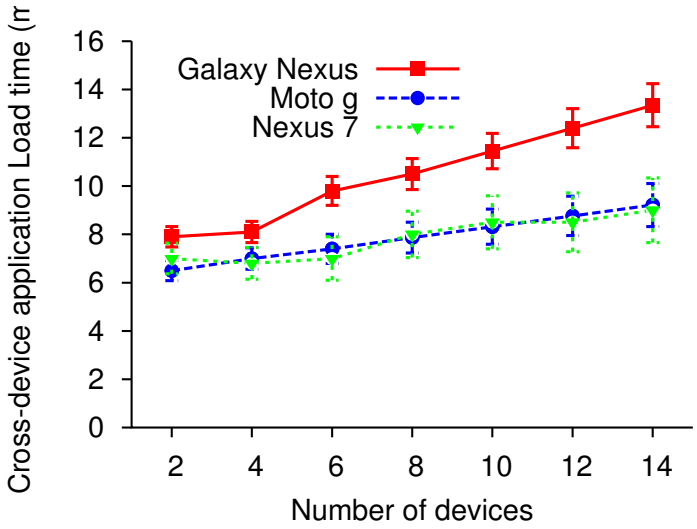


Figure 3.20: Latency of whole-home application runtime

3.8 Summary

In this Chapter we presented SPOT, a smartphone-based platform for home automation systems. SPOT addresses the extreme heterogeneity and diversity between different smart appliances in smart-home context. SPOT features a driver abstraction, in which the smart appliances are modeled by an XML-based or JAVA-based driver structure that specifies data and actions supported by appliances. By making the heterogeneous characteristics of smart appliances transparent, SPOT minimizes the burden of home automation application developers and the efforts of users who would otherwise have to deal with appliance-specific apps and control interfaces. SPOT is evaluated through several benchmarks and three case studies: cross-device programming, central usage analytics and residential energy management via demand response commands. Our evaluation demonstrates the generality of SPOT's design and its driver model.

Although our prototype and experiments focus on RESTful and SOAP-based appliances, it should be noted that the driver model and the way SPOT tackles the appliance heterogeneity are not limited to these appliances.

Chapter 4

On-device Deep Learning

4.1 Introduction

Applications of deep learning have seen a great leap in inference accuracies in a number of fields. Neural networks, the algorithmic core of deep learning, have become ubiquitous in several applications including speech recognition [Deng et al., 2013], computer vision [Moazzami et al., 2017] and natural language processing [Collobert et al., 2011]. In sensory systems, deep learning has revolutionized the way sensor measurements are processed and interpreted. However, significant requirements of memory and computational latency have been the main bottlenecks in wide adoption of these novel computational techniques on resource constrained smartphones and embedded platforms.

Deep learning models are both computationally intensive and take up a lot of srorage space, making them difficult to deploy on resource-limited embedded systems such as smartphones. For example original AlexNet [Krizhevsky et al., 2012] and R-CNN [Girshick et al., 2014] are more than 200 MB or VGGNet [Simonyan and Zisserman, 2014] is more than 520 MB. Almost all of that size is taken up with the weights for the neural connections that are trained through the training process. There are many millions of these connection in a single model. For example, VGGNet consists of more than 135 million weights in its structure [Kaparaty 2016,]. In 1998, Lecun et al. [LeCun et al.,] classified handwritten digits with less than 1M parameters, while in

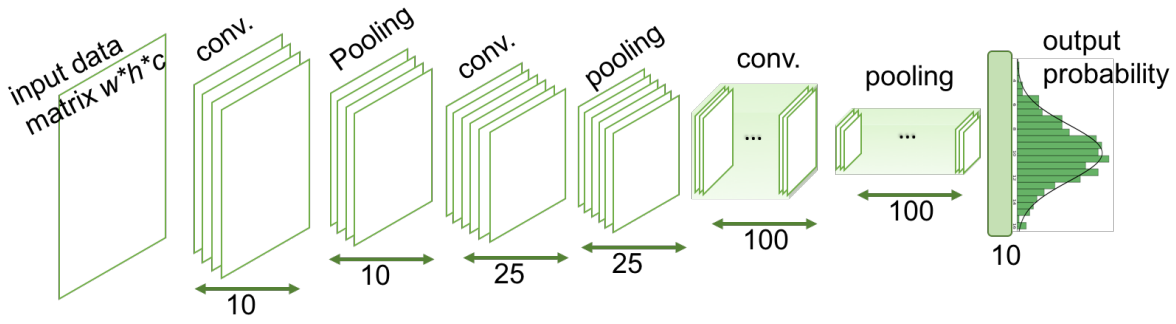


Figure 4.1: The output volume (feature maps) of different layers in a deep neural network

2012 Krizhevsky et al. [Krizhevsky et al., 2012] won the ImageNet competition with a network with 60M parameters. Deepface [Taigman et al., 2014] classified human faces with a network with 120M parameters.

These parameters are arranged in large layers; in which each layer takes input from the previous layer and after applying specific processing on the data passes its output to the next layer. Fig. 4.1 illustrates this hierarchy of layers and the input and output of each layer in a sample network architecture. The shear complexity and associated heavy computation, memory and energy demands of the deep learning models cause the majority of mobile sensor-based apps to rely on simpler methods with lower resource overhead (e.g., decision trees, Gaussian Mixture Models (GMM)) resulting in lower accuracy and robustness in real-time inference in mobile sensing applications. However, it is critical to gain the inference accuracy and robustness provided by deep models in the future generations of mobile applications.

Motivated by the above observations, in this chapter we make important strides towards adopting deep learning models by applications in mobile and embedded devices. We propose a novel model partitioning framework that enables us to embed deep learning models into mobile applications by decomposing them into their layers, and assigning each layer of the model to different tier based on the layer's time-criticality, compute-intensity, and heterogeneous latency/memory con-

sumption profiles. This is a *inter-layer* partitioning in which a neural network is partitioned in two sub-networks that are deployed on the smartphone and the cloud.

4.2 Related Work

On-device deployment by model compression: There have been various proposals to compress deep models by removing the redundancy in the weight values: Vanhoucke et al. [Gupta et al.,] explored a fixed-point implementation with 8-bit integer (vs 32-bit floating point) activations. Hwang & Sung [Shin et al., 2016] proposed an optimization method for the fixed-point network with ternary weights and 3-bit activations. Authors in [Denton et al., 2014] exploited the linear structure of the neural network by finding an appropriate low-rank approximation of the parameter and keeping the accuracy within 1% of the original model. Much work has been focused on binning the network parameters into buckets, and only the values in the buckets need to be stored. HashedNets [Chen et al.,] reduces model sizes by using a hash function to randomly group connection weights, so that all connections within the same hash bucket share a single parameter value. In their method, the weight binning is pre-determined by the hash function, instead of being learned through training.

Partitioning and task offloading: Various task offloading schemes for smartphones have been developed recently. Spectra [Flinn et al., 2002] allows programmers to specify task partitioning plans given application-specific service requirements. Chroma [Balan et al., 2003] aims to reduce the burden on manually defining the detailed partitioning plans. Medusa [Ra et al., 2012] features a distributed runtime system to coordinate the execution of tasks between smartphones and cloud. Turducken [Sorber et al., 2005] adopts a hierarchical power management architecture, in which a laptop can offload lightweight tasks to tethered PDAs and sensors. While Turducken provides

a tiered hardware architecture for partitioning, it relies on the application developer to design a partitioned application across the tiers to achieve energy efficiency. ORBIT [Moazzami et al., 2015] dispatches the execution of sensing and processing tasks in a smartphone-based multi-tier architecture to achieve *data-intensive* applications requirements. ORBIT maximizes the battery lifetime subject to the application-specific latency constraints. Wishbone [Newton et al., 2009] also features a task dispatch scheme and unlike Turducken uses a profile-based approach to find the optimal partition. It only considers two tiers: in-network and on-server. In this chapter, we take a different approach. We show how we efficiently embed deep models in mobile applications with partitioning by exploiting the model architecture and layers properties.

4.3 Architectural Observations

The layered architecture of deep models: Deep Neural Models are sequences of layers, in which each layer takes input from the previous layer and after applying specific processing on the data passes its output to the next layer. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full deep model architecture. A deep neural model can have as many of any of these kinds of layer as needed. Fig. 4.1 illustrates a typical deep neural network with several layers.

The volumetric but sparse output of each layer: Except the very first layer of a deep neural network that take advantage of kind of sensory data (e.g., sound, image, etc), the architecture of deep neural models are typically constrained in a particular way. Specifically in convolutional neural networks, the layers have neurons (a.k.a nodes) arranged in three dimensions: width, height, depth. The depth here refers to the third dimension of an activation volume or output of each layer.

In a well-trained neural network each neuron in each layer is responsible to extract certain feature from input data as a feature map which normally is a 2D matrix. As illustrated in Fig. 4.1 a 3D feature map of a layer is generated by concatenating the feature maps from all neurons in that layer. For example, CIFAR-10 model [Krizhevsky, 2009] is a deep neural model for detecting objects in tiny images. In this model the input data is image and the volume of activations has 32x32x3 dimensions (width, height, depth respectively). Obviously not all the features are available in all input data meaning many of the neurons are not activated when processing a certain input at inference time. This results in a highly sparse feature map as the output of each layer. For instance, Fig. 4.2 shows a typical-looking feature map on the first conv. layer of a trained AlexNet when it processes an image of a cat as the input. In this figure values in zero are shown in black. As we can see, the feature maps are extremely sparse since most of the activation values in the feature maps are zero. Moreover, as the data goes through the network the output volume changes. The final output layer for CIFAR-10 has dimensions of 1x1x10, because by the end of the architecture, the model reduces the input into a single vector of class scores.

Smartphone's architecture Smartphones have several salient advantages that make them promising system platforms to execute the applications with embedded deep learning models. These features include high-speed multi-core processors that are capable of executing advanced data processing algorithms, various integrated sensor modalities and multiple network interfaces.

Motivated by these observation we propose a novel partitioning mechanism that exploits the above properties in deep learning models and smartphone architecture to achieve the best embedded schema for embedding deep models in smartphone sensory apps.

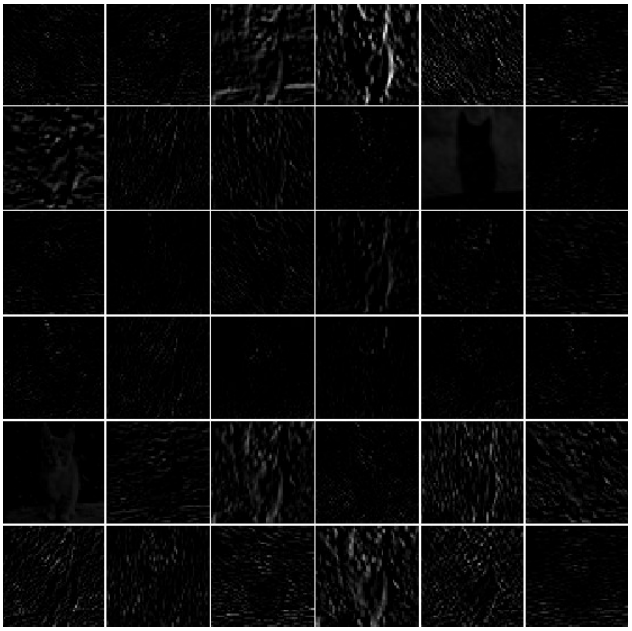


Figure 4.2: Sparsity in feature maps in conv. neural nets: This figure shows a typical-looking feature map on the first conv. layer of a trained AlexNet while processing an image of a cat as the input. Every box shows an activation map corresponding to a filter. This figure shows how sparse the activations are (most values are zero and shown in black) [Kaparaty 2016,]

4.4 Partitioning

This section describes our model partitioning framework, Deep-Partition. Deep-Partition aims to minimize the end-to-end delay of *feed-forward execution* of deep neural network on the smartphone and embedded devices. Deep-Partition employs a novel *model partitioning* framework that processes the deep neural architecture as a computation graph and finds the optimal graph cut for each model.

Consider a deep neural network model, $\mathbb{N}$, consisting of n layers (denoted by $L_1, \ldots, L_n$), with a feed-forward execution pipeline expressed as a sequential set of layers: $\mathbb{N} := L_1 \rightarrow L_2 \rightarrow \ldots \rightarrow L_n$. Let I_i denote the execution tier of L_i, where: $I_i \in \{(1,0),(0,1)\}$ represent the smartphone and cloud, respectively. Let τ_p, τ_c and τ_A denote the feed-forward execution time on the smartphone, on the cloud, and model's end-to-end, respectively. Fig. 4.1 illustrates a typical

deep learning model (a deep convolutional neural network) with hierarchy of layers and the tensor sizes for each layers output a.k.a layers activations.

We now formulate the model partitioning problem for a feed- forward execution.

Model Partitioning Problem. *For the feed-forward execution* $\mathbb{N} = L_1 \rightarrow L_2 \rightarrow \ldots \rightarrow L_n$*, the Model Partitioner finds a graph cut defined by the set* $S = \{I_1, I_2, \ldots, I_n\}$ *to minimize the total execution time in a feed-forward execution denoted by* T *subject to the processing constraints specific to each model.*

The execution times of layer L_i on the smartphone and cloud are denoted by t_i^{p} and t_i^{c}, respectively, where the superscripts 'p' and 'c' represent smartphone and cloud.Denote $t_{\mathrm{p}\leftrightarrow\mathrm{c}}$ the latency of downloading/uploading a data unit (a layer's output feature-map as a tensor with dimensions w_i, h_i, c_i) from/to the phone to/from the cloud. As we will see later in the evaluation section, unlike other partitioning schema, this parameter is very critical in defining the optimal partition due to 3D volume output of each layer as described in Section 4.3. Thus, let κ_i denote the size of output tensor of layer L_i, as $s_i = w_i \times h_i \times c_i$. In addition, let $\mathbf{K}_i$ indicate the size of this tensor considering the coding schema. For example, if the tensors are stored like images in the format of RGBA-8888 each pixel/item in the tensor is stored/computed as a 32 bit float number. Therefore $\mathbf{K}_i = 32 \times \kappa_i$. In order to keep the generality let b denote the coding bit rate (e.g., 32 for RGBA-8888) and thus $\mathbf{K}_i = b \times \kappa_i$ for the tensor output of layer L_i.

We now analyze the processing delay of a deep model in a feed-forward period.

Processing execution delay: Let $\mathcal{T}$ denote the end-to-end delay in processing the input data. Analysis shows

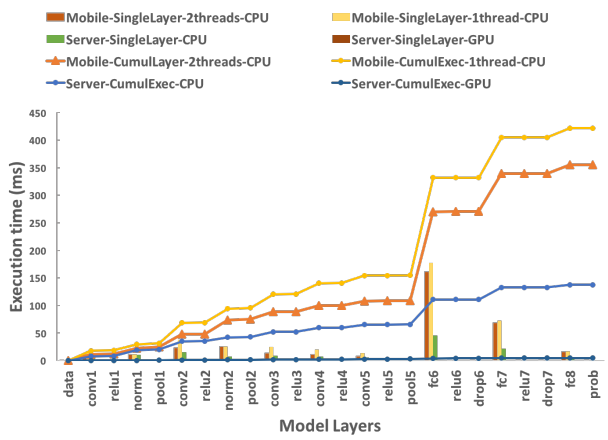
$$\mathcal{T} = \sum_{i=1}^{n} I_i \cdot t_i^{\mathrm{p}} + (1 - I_i) \cdot t_i^{\mathrm{c}} + d(I_i, I_{i-1}) \cdot \frac{1}{\lambda} \mathbf{K}_{i-1}, \tag{4.1}$$

where, similar to what we have in Chapter 2 , the function $d(I_i, I_{i-1})$ accounts for the data-copy overhead between the tiers (i.e., upload the tensor output of layer L_i to the cloud) by indicating the distance between the positions of '1' in I_i and I_{i-1}. Note that unlike Ch.2 we only consider two tiers here (smartphone and cloud) the function $d(.,.)$ can simply be defined as the XOR value of the assignments of two consecutive layers, $d(I_i, I_{i-1}) = I_i \oplus I_{i-1}$, therefore it can be either 0 or 1 depending on whether the partitioning happen between layers L_{i-1} and L_i.

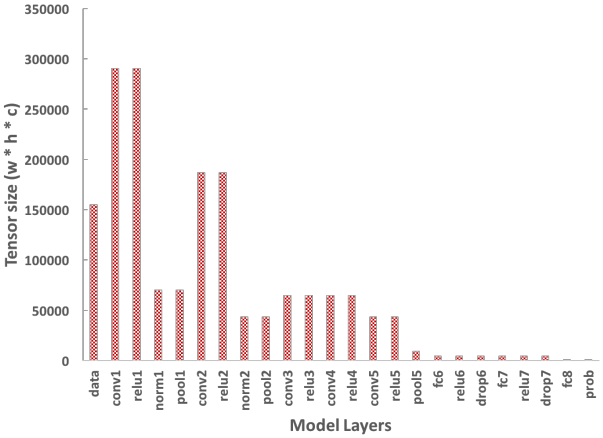
As we see under this analysis, the actual model execution time (and thus the partitioning result) not only depends on each layers computation time but also depends on the layers output volume. Parameter λ is the network's upstream bit-rate.

4.5 Layer-wise Profiling of Representative Deep Networks

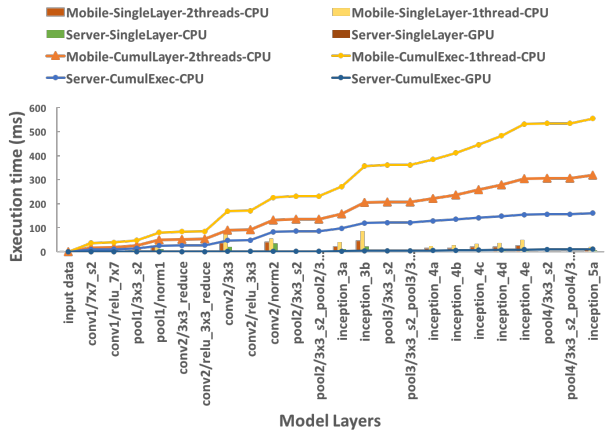
For an efficient partitioning, it is important to understand the characteristics of the execution latency in deep learning models and their internal volumetric outputs. This section presents a measurement study of the layer-wise output volume and the latency of a few of the most common representative deep models on different smartphones. The measurement study provides insights into the computational intensity of deep neural models, their layer-wise delay and output size; and motivates the key design decision in our model partitioning schema. We investigate the latency of each layer for three of the representative and commonly used deep neural models, namely AlexNet, GoogleNet and VGGNet, when running on a Galaxy S7 smartphone. Fig. 4.3a, 4.3c and 4.3e summarize these results. Moreover, Table 4.1 shows the breakdown of the output volume and execution delay for the sample neural network shown in Fig. 4.1. In addition, table 4.2 summarizes the end-to-end feed-forward execution time for these architectures.



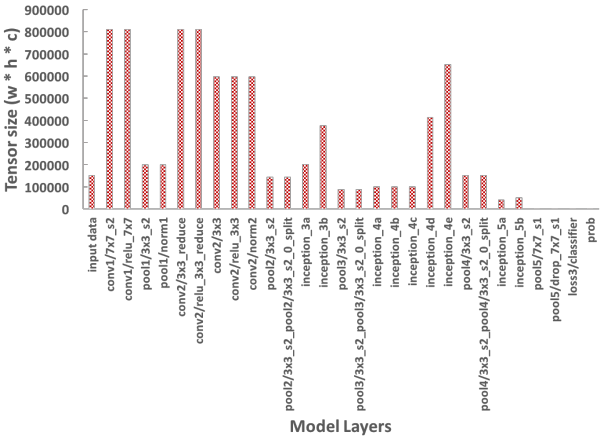
(a) AlexNet execution time



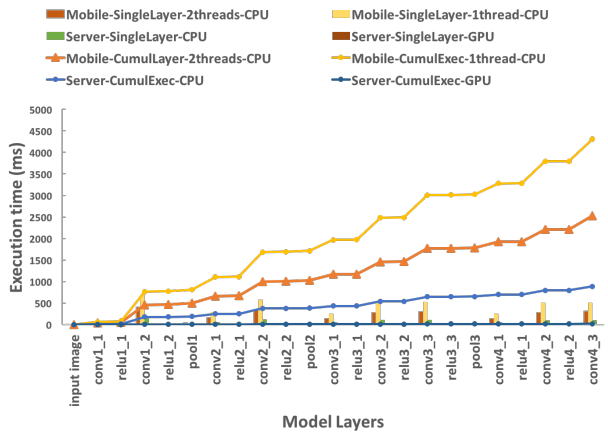
(b) AlexNet tensor size



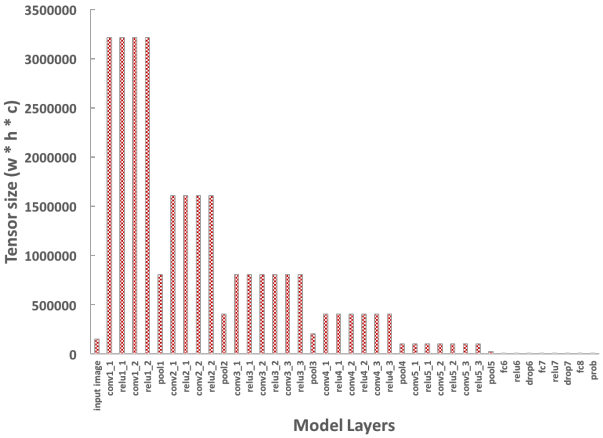
(c) GoogleNet execution time



(d) GoogleNet tensor size



(e) VGGNet execution time



(f) VGGNet tensor size

Figure 4.3: The profiling of the execution time, layer-wise latency and the activation's tensor size for the three major representative deep neural models.

Table 4.1: The breakdown of the model shown in Fig. 4.1. Each layer's output dimension and execution time profile on two different smartphones with two different processors i.e., Exynos 7420, Intel i7-4500.

	dimensions(pixel)[d]			size		execution time (ms)	
layer name	w	h	c	pixels:w*h*c	bits[β]	1.5GHz[c1]	1.8GHz[c2]
input image	32	32	3	3072	98304	NA	NA
conv1	28	28	10	7840	250880	45.56	9.39
pool1	14	14	19	3724	119168	30.66	30.49
conv2	10	10	25	2500	80000	80.86	35.29
pool2	5	5	25	625	20000	71.46	20.54
conv3	2	2	100	400	12800	50.07	21.2
pool3	1	1	100	100	3200	42.34	17.55
output	1	1	10	10	NA	NA	NA

[d]: w, h, c: indicate width, height and number of channels of the associated tensor, respectively.

[β]: Based on 32-bit RGBA-8888 coding.

[c1]: Exynos 7420, [c2] : Intel i7-4500

4.6 Evaluation of Model Partitioning

To demonstrate the expressivity of Deep-Partition and its generality and flexibility, we consider three representative Deep Convolutional Neural Networks (ConvNet) from the above list. As we discussed earlier (cf. 4.3), each ConvNet consists of different layering architecture with different intermediate feature volumes. Our goal is to demonstrate the capabilities and effectiveness of Deep-Partition rather than comparing the performance of these ConvNets.

We evaluate the effectiveness of the model partitioning algorithm presented in Section 4.4,

Table 4.2: Representative Deep Neural Network Models

Representative Deep Neural Network	Architecture$^{\alpha}$	End-to-end latency(ms)			
		Smartphone		Server	
		1-thread	2-threads	CPU-only	CPU+GPU
AlexNet	5(CV) 3(FC) 1000(O)	450	350	100	15
GoogleNet	5(CV) 3(FC) 1000(O)	500	300	150	12
VGGNet	5(CV) 3(FC) 4(PL) 1000(O)	400	220	50	10

α: FC: Fully-connected Layer, CV: Convolution Layer, PL: Pooling Layer, O: Output (number of classes)

by comparing the following partitioning baselines: a) smartphone-only, when the entire model is deployed and running on the device, and b) server-only, when the model is running on the backend server. For the execution on the smartphone we consider two cases: a) when each layers computation is performed with single thread b) when the layers computations are with two threads. Similarly, for the execution on the server side two cases are considered: a) when the computation happens on on the CPU, b) when computation happens on the GPU.

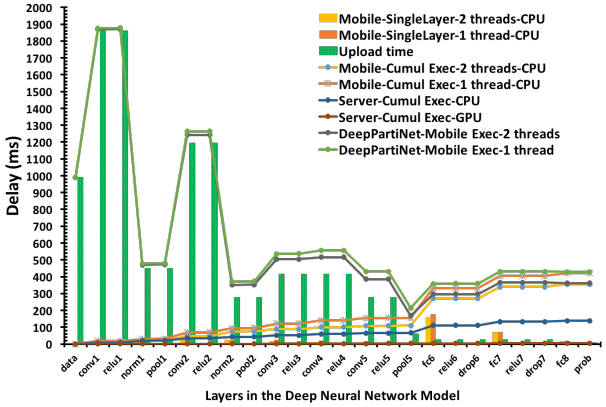
Fig. 4.3a, 4.3e and 4.3c plot the measured execution time of AlexNet, VGGNet and GoogleNet models respectively. These figures show the execution time of these models on different platforms and different configuration for each platform. Moreover, Fig. 4.3b, 4.3f and 4.3d show the output/activation volume of each layer for these models. We can see in these figures that how the model architecture and in particular the layers parameters impact the run-time performance of the models. As discussed in section 4.4 the objective of Deep-Partition is to minimize the model's latency (feed-forward execution time). On the other hand, Fig. 4.4a, 4.4e and 4.4c show the partitioning results of these models respectively. As we can see Deep-Partition suggests different partitioning layer for different models corresponding to the lowest execution time for each model.

In particular, Deep-Partition suggests partitioning AlexNet at layer *pool5* and offload the layers *fc6* onward and partitioning VGGNet at layer *pool3* and offload the execution of layers *conv4_1* to the cloud while it suggest partitioning GoogleNet at layer *pool13x3_s2* and continue the execution of the network from layer *pool13x3_s2* on the cloud.

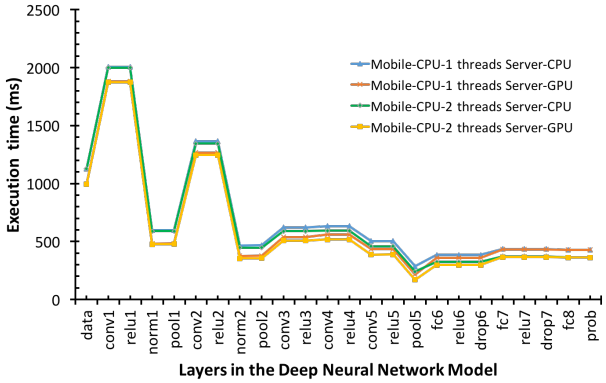
Therefore, there are three major factors that impact the partitioning results. First, the execution time of each layer on each platform as shown in the layer-wise breakdowns in Fig. 4.3a. Second, the communication delay between the smartphone and the cloud server. For example whether it is a broadband 4G communication a WiFi connection what the upstream bit-rate of the communication is. Third, the output volume of the layer's in the model. The two latter together indicate the time it takes to upload the tensor output at the edge of partition. It's easy to see that smaller tensor volume and the higher bit-rate leads to faster uploading and thus a better partitioning. As we can see the partitioning algorithm in Deep-Partition finds the minimum execution time for all the models subject to the above three factors.

In order to see the end-to-end execution time of the models that is the total execution time on the smartphone and the cloud including the latency due to uploading we plot Fig. 4.4b, 4.4d and 4.4f . The figures show the total execution time of these models versus the partitioning layer for four different configuration platforms. As we can see in these figures, Deep-Partition suggests minimum execution time for all the models.

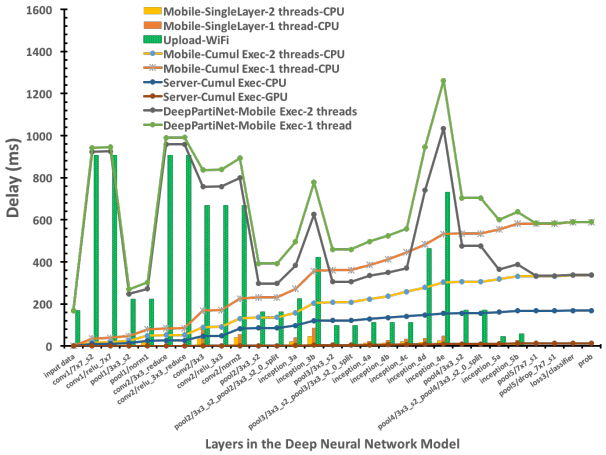
An important and interesting observation from the measurement and the results shown in figures 4.3 and 4.4 is that a few of the output tensors especially for the convolutional layer are very volumetric. This is critical for Deep-Partition partitioning decision because larger tensors faces longer delays to upload. As we cab see in Fig. 4.4 the upload time (green bars) are the dominating factor in the latency. This changes the decision of Deep-Partition in favor of layers with smaller output tensor sizes, i.e., pooling layers or fully connected layers toward the end of the network re-



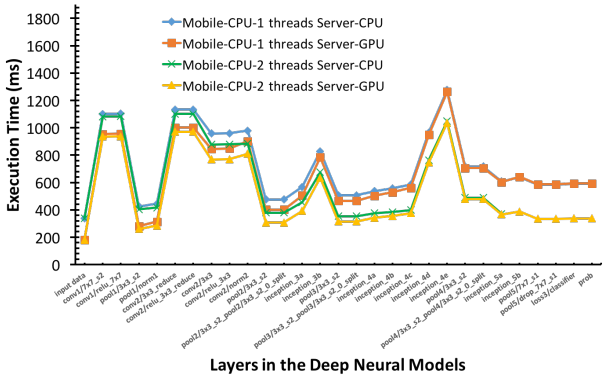
(a) AlexNet



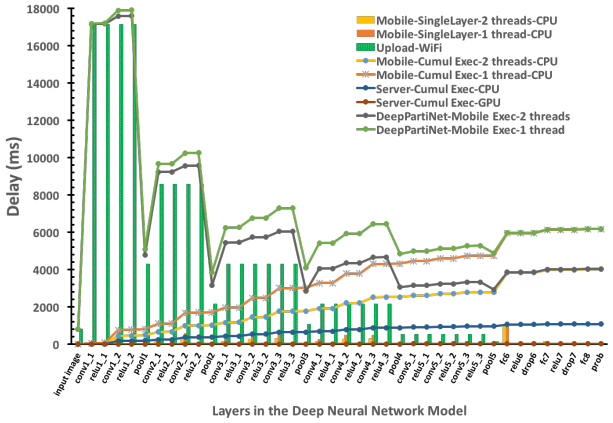
(b) AlexNet



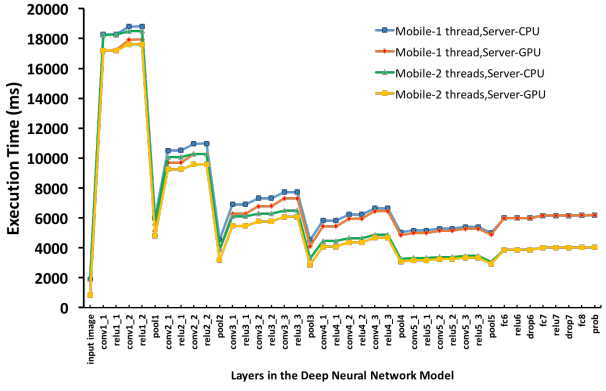
(c) GoogleNet



(d) GoogleNet



(e) VGGNet (VGG16)



(f) VGGNet (VGG16)

Figure 4.4: Partitioning Results and end-to-end model latency for representative models when Deep-Partition is applied

sulting most of the the layers assigned to the phone and thus a larger execution time on the phone. Therefore, in order to optimize the situation we consider two major factors:

The Impact of , λ, the cloud-phone communication bit-rate: Although uploading the volumetric tensors generated by the intermediate hidden layers in deep models takes longer time relatively when compared with the layers computation delay, higher communication rate can decrease this time and decrease the dominating factor of the outputs tensor sizes. Therefore, we evaluate the communication data rate (up-stream) and the impact of it on the decision made by the Deep-Partition. Fig. 4.5 shows model end-to-end latency when partitions at each layer for different communication data rates, R, for AlexNet model (without loss of generality and because of the space limitation we only show this result for AlexNet). As we see, by increasing the data rate not only the total execution time decreases (obviously), but also we see at $R = 25Mbps$ the partitioning results changes from layer *fc6* to *pool1*.

The Impact of, $||K||_0$, layer's output sparsity: Another important observations about deep neural models is that the intermediate features (feature maps) generated by hidden layers are pretty sparse. Higher sparsity means the layers output is subjects to higher compression. Similar to the communication bitrate, higher compression in the layers output decrease the dominating factor of the outputs tensor sizes in the partitioning results. Fig. 4.6 illustrates how the partition layer shifts back to the *norm1/pool1* from the *norm2/pool2* in the AlexNet architecture.

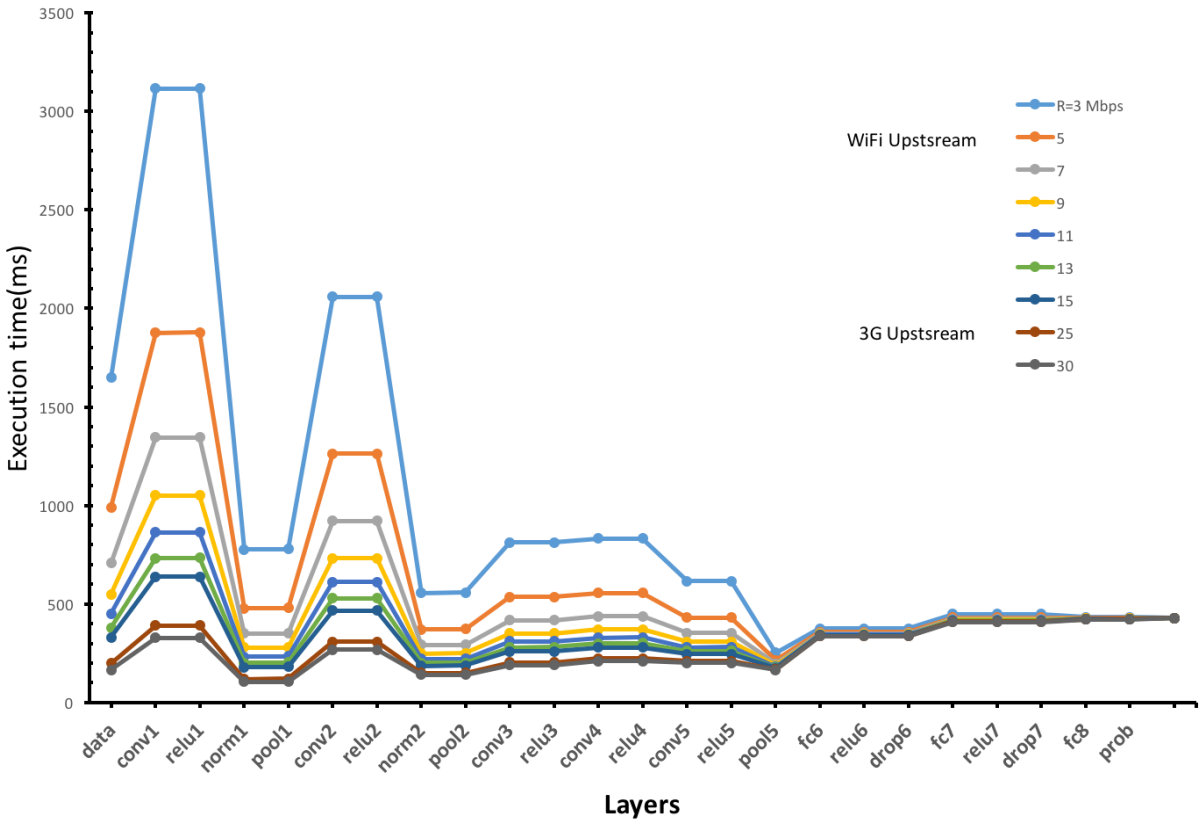


Figure 4.5: The impact of communication bit-rate on the partitioning and model latency

4.7 Application Use-case: Deep-Learning Based Crowd-Assisted Location Labeling System

In this section we take a mobile crowd-sourced based application that we prototyped as a usecase application of deep learning in mobile sensing applications. We describe *Deep-Crowd-Label*, a deep-learning based crowd-assisted location labeling system. Deep-Crowd-Label is motivated by the idea that a vast majority of location-based applications desire the semantic labeling i.e., coffee shop. Past work in location-based systems has mostly focused on achieving localization accuracy, while assuming that the translation of physical location to semantic labels will be done manually. In this section, we explore an opportunity for automatic labeling of user's location. We propose a

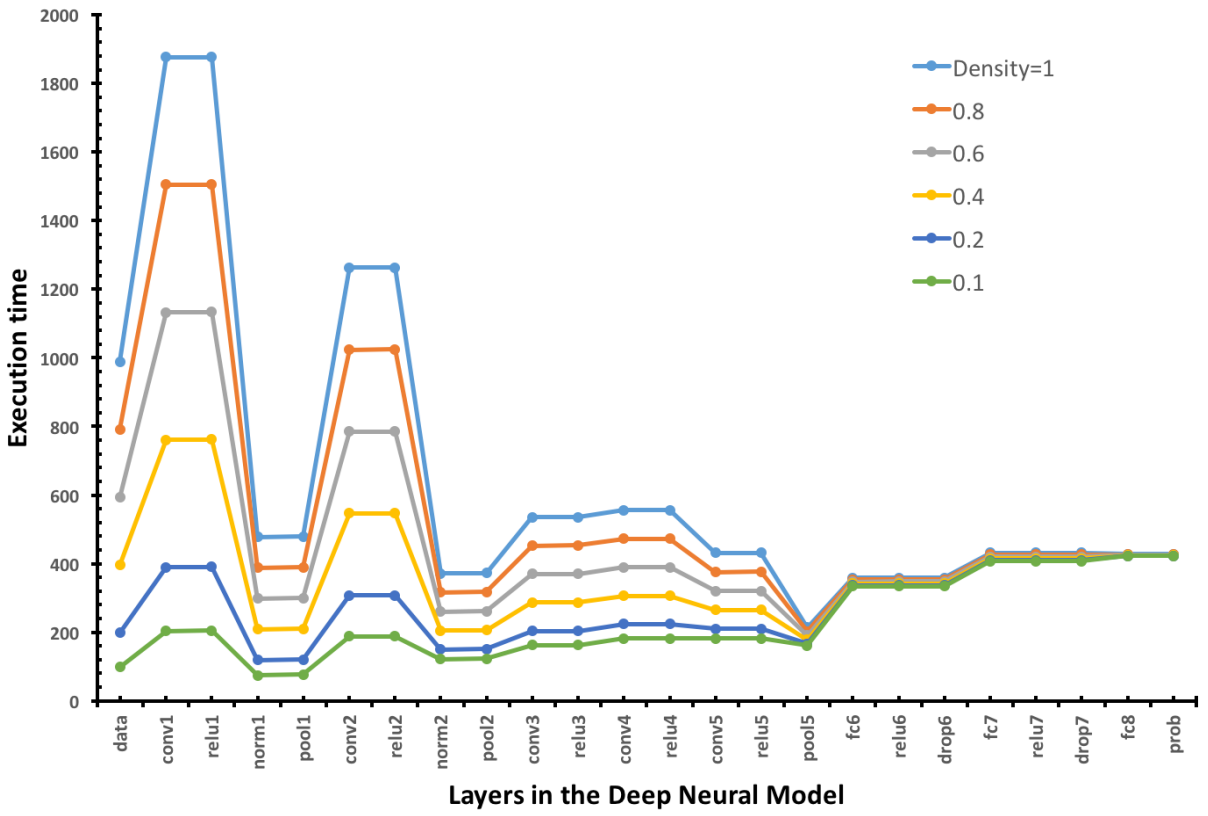


Figure 4.6: The impact of feature maps sparsity on the partitioning and model latency (AlexNet)

system called Deep-Crowd-Label that uses crowd-sensing to obtain data and build a powerful and scalable prediction model using deep neural networks. It applies convolutional neural models to predict the semantic label for the user's location. Deep-Crowd-Label also uses the power of the crowd to aggregate the prediction of the model on the data samples associated to the each location.

Prior work has shown how places can be discovered from temporal streams of user location coordinates [Chon et al., 2012]. However, if we can automatically characterize places by linking them with attributes, such as place categories (e.g., clothing store, restaurant), we can realize powerful location- and context-based scenarios [Pei et al., 2013]. Indoor maps and semantic understanding of a place are equally crucial pieces of the bigger localization puzzle, but only recently researchers began to focus on these aspects. *Deep-Crowd-Label* aims for semantically labeling

the user's location. Despite the generality of our approach we focus on the processing of indoor locations, in which the people spend a big portion of their time. In our approach we focus on the location-tagged visual data i.e., images and videos, and use deep neural networks as the core of our processing pipeline. Note that the choice of deep neural networks is due to the promising performance of the recent methods developed in this area that outperform the other data mining and machine learning methods as mentioned earlier in this chapter. Example methods include the recent algorithms developed for object detection [Zhou et al., 2014, Sharif Razavian et al., 2014, Johnson et al., 2015] or for processing of sequential information [Kelly and Knottenbelt, 2015, Pichotta and Mooney, 2016]. On the other hand, the advantages of crowd-sourcing architecture in this work is two folded. First, the training data used to train the deep models are crowd-sourced data (crowd-sensing). Second, not only in the training time, but also in our method the final label of the user's location in the inference time is aggregated over the predictions resulted from the processing of several data samples obtained by the crowd (crowd-prediction). In particular, Deep-Crowd-Label leverages the crowd-sensed data to build a big dataset suitable for training deep neural network, and proposes novel techniques based on *model adaptation* and *model extension via transfer learning* to exploit the pre-trained models to build several robust and accurate prediction models for semantically labeling the user's location. Since training deep neural networks require a lot of data to prevent over-fitting [Bishop, 2001], Deep-Crowd-Label retrofits the pre-trained models via novel transfer learning techniques and builds ensemble of models to cope the over-fitting problem. Moreover, Deep-Crowd-Label uses the power of crowd to aggregate the individual predictions to generate the final location label.

Figure 4.7: An indoor area with semantic labels

4.7.1 Traditional Approaches to Location Labeling

The notion of semantic localization is not new. Works like SurroundSense [Azizyan et al., 2009] and SenseLock [Kim et al., 2010] utilize sensor data from smartphones to characterize the ambiance and translate them into a semantic information about the user's location. Authors in [Sapiezynski et al., 2015] and [Wind et al., 2016] attempt to categorize places by training a model on WiFi, GSM and sensor data collected from frequently visited places. These approaches are based on the assumption of availability of labeled ambiance data. Several other works attempt automatic place identification (e.g., home, office, gym) based on the analysis of user trajectories, frequency and timing of visits [Krumm and Rouhana, 2013, Liu et al., 2006]. Work by [Chon et al., 2013] tries to connect the text in the crowd-sensed pictures with the posts in social networks to infer business names. AutoLabel [Meng et al., 2015] aims to automatically identify the name of the store by correlating the words inside the store's WiFi-tagged pictures with the keywords found in the store's website to produce a WiFi AP to StoreName table. Perhaps more closer to our approach is work by [Zamir et al., 2013] that attempts to identify the stores that appear in a photo by matching it against the images of the exteriors of the nearby stores extracted from the web. This approach relies

on the conventional computer vision techniques and are neither scalable nor robust. In contrast, on the one hand, Deep-Crowd-Label does not rely on the conventional computer vision techniques and uses deep neural networks that has recently driven remarkable performance in computer vision and machine learning. On the other hand, Deep-Crowd-Label relies on crowd-sourcing on both training and inference phases that increases the generality and robustness of the system.

4.7.2 Deep Learning-based Approach

Many shortcomings of traditional sensor data processing in modeling the context data can be overcome through the use of deep learning; and have been successfully applied, e.g., image captioning [Johnson et al., 2015] or time series data [Kelly and Knottenbelt, 2015]. Such deep algorithms (e.g., CNN, RNN) learn a number of hierarchical layers of dense feature representations and has two important benefits. First, deep neural networks do not need hand-crafted features which allows us to deploy the models in real applications [LeCun et al., 2015]. Second, the deep features learned by deep neural networks are more generic and extremely robust which allows us to use the learned features in one domain (the source domain) to build the models in another domain (the target domain), e.g., object detection to scene understanding. In this section we present our deep learning approach and the design of our processing pipeline.

Domain Adaptation with Pre-Trained Models

Domain adaptation aims at training a classifier in one problem space and applying it to a related but not identical problem [Zhang et al., 2015]. We adopt existing practice in DNN modeling, called pre-trained models [Jia et al., 2014], and apply them to our location labeling problem. Our “domain adaptation" in this case is limited to the “label space adaptation", that is adopting the output of the

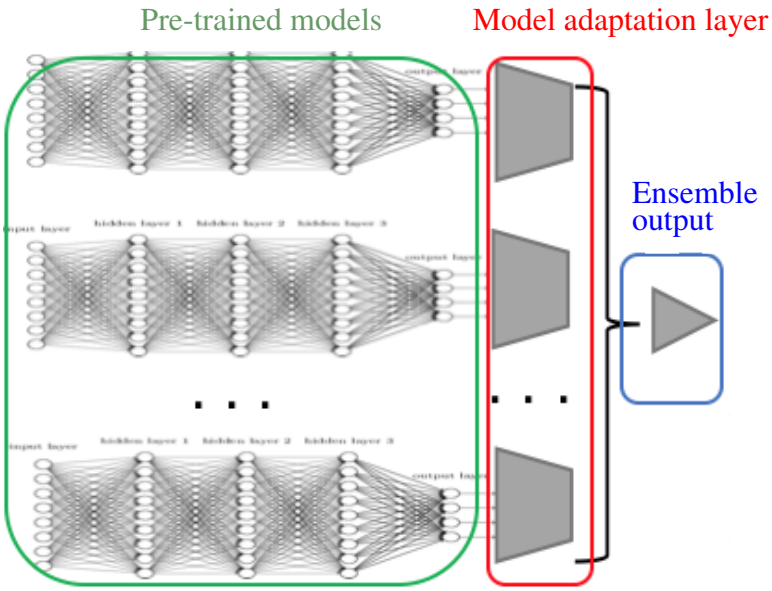


Figure 4.8: Our model adaptation schema and ensemble of adapted deep neural models. Left/Green: Several deep neural models pre-trained or extended using transfer learning. Middle/Red: The adaptation layer. Right/Blue: The aggregation layer.

final layer without tuning the learned parameters (weights) or the internal network structure. In this approach, the pre-trained models trained with arbitrary number of class labels used in the task of classifying the images that cover only a subset of classes identifying the location context i.e., store types. Table 4.3 summarizes the pre-trained models used in our domain adaptation mechanism. The final layer of these DNNs commonly use the SoftMax classifier [Bishop, 2001]. For this layer we have:

$$P(y = j|X_i) = \frac{e^{w_j X_i^T}}{\sum_{k=1}^{n} e^{(w_k X_i^T)}} \tag{4.2}$$

where X_i is the feature vector extracted by the deep neural network for the input sample i (captured single image). w_i is the weight learned by the neural network. y is the predicted class label in $j \in \mathsf{N}$ the set of all the class labels a pre-trained model is trained on (the source domain). For example the size of label space, $|\mathsf{N}|$, for original AlexNet [Krizhevsky et al., 2012] trained for ImageNet [Deng et al., 2009] is 1000 labels. In order to adapt such a pre-trained model for the task of interest (the

target domain), we follow the Bayesian chain rule [Bishop, 2001] and apply the prior knowledge specific to the space to the model prediction and thus we have:

$$P_s(y = j|X_i) = \frac{1(y \in \mathbb{L}) \cdot P(y = j|X_i)}{\sum_{\mathbb{L}} 1(y \in \mathbb{L}) \cdot P(y = j|X_i)} \tag{4.3}$$

where $1(.)$ is the identity function and $\mathbb{L}$ is the label-set of the application the pre-trained model is adopted for (the target domain). The denominator is the normalization factor and thus $P_s(y = j|X_i)$ indicates the probability of class(label) given the feature vector X_i for application specific labels $j \in \mathbb{L}$. Therefore, given a pre-trained model $\mathbb{M}$ with the label space $\mathbb{N}$ in the source domain, and the loss function as shown in equation 4.2, our domain adaptation approach adapts the model $\mathbb{M}$ for the target application with label space $\mathbb{L} \subset \mathbb{N}$ using the equation 4.3. Fig. 4.8 illustrates the layouts of our pipeline. The adaptation layer in this figure implements equation 4.3.

Model Extension with Transfer Learning

In practice, very few people train an entire deep neural network from scratch, because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pre-train a deep neural net. on a very large standard dataset i.e. ImageNet, which contains 1.2 million images with 1000 categories or Places dataset [también SCHUM,]] with 500K images with 205 categories and then use the the resulting model (the pre-trained model) either as an initialization or a fixed feature extractor for the task of interest i.e. location-context/scene understanding. In the previous section, we described how this models are adapted to our application without further training. This approach works perfect for the cases that the target labels are a subset of original labels, $\mathbb{L} \subset \mathbb{N}$. However, we found out, in our problem, none of the original models include the entire label space. Therefore, we have two case: a) There are class labels in $\mathbb{L}$ that do not have any high level representation

Table 4.3: Models built in Deep-Crowd-Label via model adaptation and model extension

DNN Model	Architecture$^{\alpha}$	Dataset (# of classes/ total size)	M$^{\mathfrak{M}}$
imagenet-alexnet	5(CV), 3(FC), 1000(O)	ImageNet(1000/1.2M)	MA
places-alexnet	5(CV), 3(FC), 205(O)	Places (205/2.5M)	MA
places-hybrid	5(CV), 3(FC), 1183(O)	ImageNet (978) + Places(205)	MA
places-googleNet	59(CV), 5(FC),205(O)	Places (205/2.5M)	MA
shops-alexnet	5(CV), 3(FC), 26(O)	Places205 $\Longrightarrow$ shops from places + SUN397[1] dataset (205/2.5M)	ME
indoor67-alexnet	5(CV), 3(FC), 67(O)	Indoor 67[2] (67/15.6K)	MA
indoorshops-alexnet	5(CV), 3(FC), 26(O)	ImageNet $\Longrightarrow$ indoor shops from SUN + places $\Longrightarrow$ data collected by ourselves (15/1.5k)	ME
imagenet-stores-alexnet	5(CV), 3(FC), 9(O)	ImageNet$\Longrightarrow$ indoor shops from ImageNet (9/10k)	ME

α: FC: Fully-connected Layer, CV: Convolution Layer, O: Output (number of classes)

$\Longrightarrow$: indicates the direction of transfer learning: base model $\Longrightarrow$ new model.

M$^{\mathfrak{M}}$(Method) : MA: Model Adaptation (Sec. 4.7.2), ME: Model Extension (Sec. 4.7.2).

in the pre-trained models label space $\mathbb{N}$ e.g., computer store is missing in the ImageNet label space. b) There are class labels in $\mathbb{L}$ that match with more than one class label in $\mathbb{N}$ e.g., shoe-shop has multiple corresponding categories in AlexNet-ImageNet model (shoe, loafer shoe or sport shoe). To address this issue we propose a“Transfer Learning" [Pan and Yang, 2010] approach. In this approach we keep the feature extractor layers[3] in the pre-trained model frozen by setting the

[3] convolutional layers

learning rate for these layers to zero. The last fully connected layer instead is initiated with random weights and is trained with the data collected and labeled by us. This allows us to use the features extractors trained in the pre-trained models and use the our collected data to train the final fully connected layers to cover the entire label space $\mathbb{L}$. This enables us to train a deep model with limited amount of training data with no over-fitting.

Model Ensemble

To further improve the accuracy and increase the robustness of our final label prediction model (for single image) we use an ensemble of models as illustrated in Fig. 4.8. Our ensemble model is simply the weighted average of prediction probabilities of the individual models that are either adapted or extended by the methods explained earlier (cf. Sec.4.7.2 4.7.2). Fig.4.8 illustrates this approach and Table 4.3 summarizes these models.

4.7.3 Training

We train our network with a combination of available public datasets and our collected data using video frames and still images. The training is done when the Model Extension (ME) approach (cf. Sec. 4.7.2) is used. Each model in this approach is trained independently regardless of what kind of pre-trained model is chosen as the base model. In the training process, the convolutional layers are taken from the convolutional layers in the base model e.g., [Jia et al., 2014, Krizhevsky et al., 2012]. We pass the output of these convolutional layers (i.e. the pool5 features) into a single feature vector. This vector is the input to the fully connected layers taken from the base model and trained on our dataset. Finally, we re-define the last layer (i.e., softmax layer) to have outputs equal to the number of classes in our label space. During the training procedure the learning rate is set to zero for the convolutional layers. This is because we do not have enough data to train these

layers and thus by freezing these layers (learning rate = 0) it will prevent our model to over-fit to the training data and at the same time taking advantages of the pre-trained model as a feature extractor. The learning rate for the fully connected layers are taken from the default for the base layers while the learning rate for the output layer is set to 10 times of the maximum of learning rates of the fully connected layers. This is because the last layer defined specifically for this layer, unlike other fully connected layers there is no corresponding layer in the base model and thus the weights are initialized randomly. Therefore this requires us to have the layer trained faster than other fully connected layers. Beside the learning rate and the structure of the output layer, the other network hyper-parameters are taken from the base models. In our model we use ReLU [Nair and Hinton, 2010] as the activation function in each fully-connected and dropout [Srivastava et al., 2014] after each one, as in the base models. Our neural network is trained using Caffe [Jia et al., 2014].

4.7.4 Labeling and Aggregation by Crowd-Sourcing

The last stage of our location-labeling pipeline is to aggregate the prediction results of individual images associated to one location, k. We do not need to perform this step at the training time but only at the inference time, that is when the trained model is used to label the collection of images from a location. Let $P_I(y = j|X_i)$ be the prediction probability of classifying an image feature vector X_i using our ensemble of deep neural network models (cf. Sections 4.7.2), and let Γ_k indicate the set of images collected for location k . Therefore we have:

$$P_\Gamma(y = i|\Gamma_k) = \frac{1}{|\Gamma_k|} \sum_{x_i \in \Gamma_k} P_I(y = j|X_i)$$

where $P_\Gamma(y = i|\Gamma_k)$ is the aggregated predication for each location k. The system labels each

location by picking the label with the maximum prediction probability, in other words we have:

$$label_k = \max_l P_\Gamma(y|\Gamma_l) \tag{4.4}$$

4.7.5 Data Collection and Dataset Preparation

Data Collection: We have collected data from 26 different indoor locations, mostly shops in the malls and supermarkets. The data is collected using smart-watch and smart-phone in the form of videos and still images. The videos are converted to the frames. It is important to remove the very similar frames in the training data to prevent bias in the model. Therefore we only extract the key-frames from the video using FFMPEG. Moreover, the 80% data is left for training and the 20% of is used for inference (labeling). Since having a balanced dataset is crucial in the training phase, the number of images per class is kept balanced in the training set. This is not necessary in the inference phase.

Automatic Rotation and Noise Reduction: During data collection we observed that the camera API on each device rotates the captured image arbitrary. Since our deep neural network models are not rotation-invariant, we use the camera information in the images Exif meta-data, to rotate the images to the right orientation automatically. In addition, we apply standard denoising method to improve the quality of collected images. Images that the measure of blurriness is above a certain threshold are not used in the training data.

Data Augmentation: Deep neural networks require a lot of data to train. The easiest and the most common method to cope with data scarcity is to artificially enlarge the dataset a.k.a as data augmentation. Following the technique in [Krizhevsky et al., 2012], we augment our dataset by extracting random 224×224 patches (and their horizontal reflections) from the 256×256 images

and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent [Krizhevsky et al., 2012]. Without this scheme, our network suffers from substantial over-fitting even with transfer learning (cf. Sec. 4.7.2).

4.7.6 Evaluation

Our proposed method is applied to the real data collected from 26 stores. Fig. 4.9 shows the prediction results of our pipeline when it labels a single image. The figure shows top-5 prediction results for each image with confidence values in the bar chart in increasing order. Moreover, Tables 4.4 -a:f show the results of the aggregated predictions for 6 different indoor locations (5 different kinds of stores and 1 food court) in a shopping mall. For each location the grand-truth label is mentioned on the top-right cell and the top-5 prediction results are reported in descending order of confidence values. Although the confidence values are different, the results show the difference between the top-1 prediction the other 4 verifying the applicability and generality of our method in predicting the right label for each location. Moreover, as we can see in Fig. 4.9 there are several examples that even the top-1 prediction is not correct (false positive) while this is not the case when the results are aggregated by crowd-sourcing as it is shown in Tables 4.4 a-f. This results show the expressivity of our method in aggregating with crowd-sensing (crowd-prediction) to improve the prediction accuracy.

Figure 4.9: Predictions on real samples collected from indoor shops. Bars below each image show the top-5 model predictions using our deep learning method sorted in ascending order.

4.8 Summary

Deep neural models are both computationally and memory intensive, making them difficult to deploy on mobile application with limited hardware resources. In this chapter we presents Deep-Partition, an optimization based partitioning pipeline featuring a tiered architecture for smartphone and the back-end cloud to deploy and execute deep neural models more efficiently. Deep-Partition provides a profile-based model partitioning allowing it to intelligently dispatch the processing tasks among the tiers to minimize the smartphone power consumption and the deep models feed-forward latency. Extensive microbenchmark evaluation and three case studies on representative deep neural

Table 4.4: Location labeling results. Each table represents one store with name and grand-truth type (top row). Top-5 prediction results with confidence values (prediction probabilities) are presented in each row. Each prediction is the aggregated result of crowd-sensed images for each store (Sec. 4.7.4).

(a)

Safeway	supermarket
68.52%	supermarket
6.31%	cottage-garden
5.06%	crevasse
4.89%	valley
4.81%	mountain

(b)

Macy's	clothing-store
35.71%	clothing-store
5.75%	gift-shop
3.99%	staircase
3.81%	shoe-shop
3.13%	beauty-salon

(c)

Disney Store	gift-shop
52.60%	gift-shop
11.97%	candy-store
5.18%	market
3.06%	game-room
2.42%	supermarket

(d)

Apple Store	computer store
16.47%	computer store
6.42%	food-court
6.38%	art-gallery
5.49%	cafeteria
5.26%	art-studio

(e)

Zara	clothing-store
40.40%	clothing-store
5.20%	garbage-dump
3.98%	slum
2.91%	excavation
2.83%	railroad-track

(f)

DSW	shoe-shop
19.15%	bookstore
11.39%	airport-terminal
7.32%	shoe-shop
6.32%	supermarket
4.86%	clothing-store

models validate the performance gain by Deep-Partition.

In addition, this chapter presents Deep-Crowd-Label, a novel system to semantically label user's location. Deep-Crowd-Label is a crowd-assisted system that uses crowd-sourcing in both training and inference time. It builds deep convolutional neural models using crowd-sensed images to infer the context (label) of indoor locations. It features domain adaptation and model extension via transfer learning to efficiently build deep models for image labeling. By fully exploiting the pre-

trained models and available datasets, Deep-Crowd-Label builds ensemble of models to increase the robustness and improve the accuracy of prediction. Moreover, Deep-Crowd-Label aggregates the several individual predictions of images obtained from the same location to infer the contextual label of a location. We also provide the layer-wise benchmark our deep models and apply novel compression techniques on the trained models to facilitate the deployment of the deep neural network on smartphones. The prototyped system and the preliminary experiments on 26 different stores show the high accuracy of the model and demonstrates the generality and robustness of the underlying approach. Future plans include extending the model to more diverse types of locations as well as improving the on-device performance.In addition we plan to merge our ensemble of models into one unified deep neural network by exploiting the shared part of the models.

Chapter 5

Conclusion

Supported by advanced sensing capabilities, increasing computational resources and the advances in Artificial Intelligence, smartphones have become our virtual companions in our daily life. An average modern smartphone is capable of handling a wide range of tasks including navigation, advanced image processing, speech processing, cross app data processing and etc. The key facet that is common in all of these applications is the data intensive computation.

In this dissertation we have taken steps towards the realization of the vision that makes the smartphone truly a platform for data intensive computations by proposing frameworks, application and algorithmic solutions. We followed a data-driven approach to the system design. To this end, several challenges must be addressed before smartphones can be used as a system platform for data-intensive applications. The major challenge addressed in this dissertation include high power consumption, high computation cost in advance machine learning algorithms, lack of real-time functionalities, lack of embedded programming support, heterogeneity in the apps, communication interfaces and programming abstractions and lack of customized data processing libraries.

The contribution of this dissertation can be summarized as follows. We presented the design, implementation and evaluation of the ORBIT framework, which represents the first system that combines the design requirements of a machine learning system and sensing system together at the same time. We ported for the first time off-the-shelf machine learning algorithms for real-time sensor data processing to smartphone devices. In this process we considered the power and mem-

ory limitation of smartphone, and for each algorithm we provided two versions: the light and the heavy version. This is a leap forward from previous approaches, which relied on custom-designed sensing and computing platforms. We highlighted how machine learning on smartphones comes with severe costs that need to be mitigated in order to make smartphone capable of real-time data-intensive processing. Some of the costs can be managed with an adapting re-design of the off-the-shelf processing pipeline with additional real-time hyper-parameter control parameters to control the precision and computation cost of the pipeline respect to available resource smartphone in terms of battery duration. We showed that some of the limitations imposed by a mobile sensing application can be overcome by having a multi-tier framework allowing us to split the computation pipeline between the smartphone and two other tiers namely extension-board and cloud, by identifying the bottlenecks in the computation graph. We showed that computation blocks can be can be adopted at execution time leading to further improvement in the resource consumption while maintaining the algorithm accuracy and yet shortening the computation time. We reported on our experience deploying ORBIT at scale with a few case studies as well as multiple deployments on active volcanos in Ecuador and Chile.

We extended the scope of our work from platforms to application and presented SPOT. SPOT aims to address some of the challenges discovered in mobile-based smart-home systems. These challenges prevent us from achieving the promises of smart-homes due to heterogeneity in different aspects of smart devices and the underlining systems. This owes to lack of dominating standards in smart-home technologies, leading to the fragmented digital homes rather than truly smart homes. We face the following major heterogeneities in building smart-homes:: (i) Diverse appliance control apps (ii) Communication interface, (iii) Programming abstraction. SPOT is an enabling technology for smart-homes system that allows the integration of hetrogenious smart-device seemless by proposing a novel dynamic draver loading schema. SPOT introduces two driver models namely

XML-based and library-based allowing the integration and manipulation of smart devices easy for both programmers and users. SPOT makes the heterogeneous characteristics of smart appliances transparent, and by that minimizes the burden of home automation application developers and the efforts of users who would otherwise have to deal with appliance-specific apps and control interfaces. SPOT is evaluated through several benchmarks and three case studies: cross-device programming, central usage analytics and residential energy management via demand response commands. Our evaluation demonstrates the generality of SPOTâĂŹs design and its driver model.

After discussing two aspects of this dissertation namely the framework and the application, we finally presented the algorithmic aspect of the dissertation by introducing two systems in smartphone-based deep learning area: Deep-Crowd-Label and Deep-Partition. Deep neural models are both computationally and memory intensive, making them difficult to deploy on mobile applications with limited hardware resources. On the other hand, they are the most advanced machine learning algorithms suitable for real-time sensing applications used in the wild. Deep-Partition is an optimization based partitioning meta-algorithm featuring a tiered architecture for smartphone and the back-end cloud, which helps to deploy and execute deep neural models more efficiently. Deep-Partition provides a profile-based model partitioning allowing it to intelligently execute the Deep Learning algorithms among the tiers to minimize the smartphone power consumption by minimizing the deep models feed-forward latency. Extensive microbenchmark evaluation and three case studies on representative deep neural models validate the performance gain by Deep-Partition. In addition, we presented Deep-Crowd-Label, a novel algorithm designed for distributed collaborative smartphone systems for crowd-sourcing applications. Deep-Crowd-Label is prototyped for semantically labeling userâĂŹs location. Deep-Crowd-Label is a crowd-assisted algorithm that uses crowd-sourcing in both training and inference time. It builds deep convolutional neural models using crowd-sensed images to detect the context (label) of indoor locations.

It features domain adaptation and model extension via transfer learning to efficiently build deep models for image labeling. By fully exploiting the pre-trained models and available datasets, Deep-Crowd- Label builds ensemble of models to increase the robustness and improve the accuracy of prediction. Moreover, Deep-Crowd-Label aggregates several individual predictions of images obtained from the same location to infer the contextual label of a location. The prototyped system and the preliminary experiments on 26 different in-door locations show the high accuracy of the model and demonstrates the generality and robustness of the underlying approach.

The work presented in this dissertation covers three major facets of data-driven and compute-intensive smartphone-based systems, platforms, applications and algorithms; and helps to spurs a new area of research on smartphone sensing and opens up new directions in mobile computing research.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Hom,] Control your home. http://www.homeseer.com/. [Online; accessed 2-Nov-2014].

[DEX,] Dalvik executable format. https://source.android.com/devices/tech/dalvik/dex-format.html. [Online; accessed 03-Apr-2015].

[sys,] Display and performance analysis in android apps. http://developer.android.com/tools/debugging/systrace.html. [Online; accessed 06-Apr-2015].

[ELK,] Elk products. http://www.elkproducts.com/security-automation-connection. [Online; accessed 2-Nov-2014].

[fas,] Fast demand response (white paper). https://www.parc.com/content/attachments/energy_fastdemandresponse_wp_parc.pdf. [Online; accessed 06-Apr-2015].

[GE,] Ge brillion appliances. http://www.geappliances.com/connected-home-smart-appliances/. [Online; accessed 2-Nov-2014].

[Gre,] Green button data. http://www.greenbuttondata.org/. [Online; accessed 4-Nov-2014].

[Con,] Home automation and control. http://www.control4.com. [Online; accessed 2-Nov-2014].

[IFT,] If this, then that. https://ifttt.com/. [Online; accessed 03-Apr-2015].

[Nes,] Nest thermostat and smoke detector. https://nest.com/. [Online; accessed 2-Nov-2014].

[upn,] Open source frameworks for upnp. http://www.cybergarage.org/do/view/Main/UPnPFramework. [Online; accessed 06-Apr-2015].

[Ope,] Openadr alliance, openadr 2.0 profile specification. http://www.openadr.org/specification. [Online; accessed 3-July-2014].

[Phi,] Philips hue light. www.lighting.philips.com. [Online; accessed 2-Nov-2014].

[Rad,] Radio thermostat. http://www.radiothermostat.com/. [Online; accessed 2-Nov-2014].

[sma,] Smart-things. http://www.smartthings.com/. [Online; accessed 2-Nov-2014].

[eco,] Smart wifi thermostats by ecobee. http://www.ecobee.com/. [Online; accessed 2-Nov-2014].

[ufo,] Ufo power center. http://www.energyufo.com/. [Online; accessed 06-Apr-2015].

[Ven,] Venstar thermostat. http://www.venstar.com/Thermostats/. [Online; accessed 2-Nov-2014].

[wem,] Wemo that – home automation made easy. http://www.wemothat.com/. [Online; accessed 29-Mar-2015].

[win,] Wink home automation system. http://www.wink.com/products/. [Online; accessed 2-Nov-2014].

[Amin et al., 2007] Amin, S., Bayen, A. M., El Ghaoui, L., and Sastry, S. (2007). Robust feasibility for control of water flow in a reservoir-canal system. In *Decision and Control, 2007 46th IEEE Conference on*, pages 1571–1577. IEEE. http://float.berkeley.edu.

[Arduino Board,] Arduino Board. Arduino board. http://www.arduino.cc.

[Azizyan et al., 2009] Azizyan, M., Constandache, I., and Roy Choudhury, R. (2009). Surroundsense: mobile phone localization via ambience fingerprinting. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 261–272. ACM.

[Balan et al., 2003] Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T. (2003). Tactics-based remote execution for mobile computing. In *MobiSys*.

[Bishop, 2001] Bishop, C. (2001). Bishop pattern recognition and machine learning.

[Chen et al.,] Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. Compressing neural networks with the hashing trick.

[Chon et al., 2013] Chon, Y., Kim, Y., and Cha, H. (2013). Autonomous place naming system using opportunistic crowdsensing and knowledge from crowdsourcing. In *Information Processing in Sensor Networks (IPSN), 2013 ACM/IEEE International Conference on*. IEEE.

[Chon et al., 2012] Chon, Y., Lane, N. D., Li, F., Cha, H., and Zhao, F. (2012). Automatically characterizing places with opportunistic crowdsensing using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM.

[Chu et al., 2011] Chu, D., Lane, N. D., Lai, T. T.-T., Pang, C., Meng, X., Guo, Q., Li, F., and Zhao, F. (2011). Balancing energy, latency and accuracy for mobile sensor data classification. In *SenSys*.

[Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

[Cuervo et al., 2010a] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010a). Maui: making smartphones last longer with code offload. In *MobiSys*.

[Cuervo et al., 2010b] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010b). Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA. ACM.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[Deng et al., 2013] Deng, L., Li, J., Huang, J. T., Yao, K., Yu, D., Seide, F., Seltzer, M., Zweig, G., He, X., Williams, J., Gong, Y., and Acero, A. (2013). Recent advances in deep learning for speech research at microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*.

[Denton et al., 2014] Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277.

[Dixon et al., 2010] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, V. (2010). The home needs an operating system (and an app store). In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 18:1–18:6, New York, NY, USA. ACM.

[Eagle and Pentland, 2006] Eagle, N. and Pentland, A. (2006). Reality mining: sensing complex social systems. *Personal and ubiquitous computing*, 10(4):255–268.

[Escoffier et al., 2008] Escoffier, C., Bourcier, J., Lalanda, P., and Yu, J. (2008). Towards a home application server. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 321–325. IEEE.

[Faulkner et al., 2011] Faulkner, M., Olson, M., Chandy, R., Krause, J., Chandy, K. M., and Krause, A. (2011). The next big one: Detecting earthquakes and other rare events from community-based sensors. In *IPSN*.

[Flinn et al., 2002] Flinn, J., Park, S., and Satyanarayanan, M. (2002). Balancing performance, energy, and quality in pervasive computing. In *ICDCS*.

[Floating sensor network project,] Floating sensor network project. Floating sensor network. http://float.berkeley.edu.

[Girod et al., 2004] Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., and Estrin, D. (2004). Emstar: A software environment for developing and deploying wireless sensor networks. In *USENIX Annual Technical Conference*.

[Girshick et al., 2014] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition*.

[Guizzo, 2011] Guizzo, E. (2011). Robots with their heads in the clouds. *IEEE Spectrum*, 48(3):16–18.

[Gumstix,] Gumstix. Gumstix. https://www.gumstix.com.

[Gupta et al.,] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision.

[Ha et al., 2007] Ha, Y.-G., Sohn, J.-C., and Cho, Y.-J. (2007). ubihome: An infrastructure for ubiquitous home network services. In *Consumer Electronics, 2007. ISCE 2007. IEEE International Symposium on*, pages 1–6. IEEE.

[Hammer-Lahav and Hardt,] Hammer-Lahav, D. and Hardt, D. The oauth2. 0 authorization protocol. 2011. Technical report, IETF Internet Draft.

[IOIO for Android,] IOIO for Android. IOIO for Android. www.sparkfun.com.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM.

[Johnson et al., 2015] Johnson, J., Karpathy, A., and Fei-Fei, L. (2015). Densecap: Fully convolutional localization networks for dense captioning. *arXiv preprint arXiv:1511.07571*.

[Ju et al., 2012] Ju, Y., Lee, Y., Yu, J., Min, C., Shin, I., and Song, J. (2012). Symphoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *SenSys*.

[Kang et al., 2008] Kang, S., Lee, J., Jang, H., Lee, H., Lee, Y., Park, S., Park, T., and Song, J. (2008). Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *MobiSys*.

[Kang et al., 2010] Kang, S., Lee, Y., Min, C., Ju, Y., Park, T., Lee, J., Rhee, Y., and Song, J. (2010). Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *PerCom*.

[Kaparaty 2016,] Kaparaty 2016. Conv net break down. http://cs231n.github.io/convolutional-networks.

[Kehoe et al., 2015] Kehoe, B., Patil, S., Abbeel, P., and Goldberg, K. (2015). A survey of research on cloud robotics and automation. *IEEE Transactions on automation science and engineering*, 12(2):398–409.

[Kelly and Knottenbelt, 2015] Kelly, J. and Knottenbelt, W. (2015). Neural nilm: Deep neural networks applied to energy disaggregation. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM.

[Kim et al., 2010] Kim, D. H., Kim, Y., Estrin, D., and Srivastava, M. B. (2010). Sensloc: sensing everyday places and paths using less energy. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 43–56. ACM.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[Krumm et al., 2000] Krumm, J., Harris, S., Meyers, B., Brumitt, B., Hale, M., and Shafer, S. (2000). Multi-camera multi-person tracking for easyliving. In *Visual Surveillance, 2000. Proceedings. Third IEEE International Workshop on*, pages 3–10.

[Krumm and Rouhana, 2013] Krumm, J. and Rouhana, D. (2013). Placer: semantic place labels from diary data. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*.

[LeCun et al.,] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation*.

[LG Optimus Net,] LG Optimus Net. LG Optimus Net. http://www.gsmarena.com/lg_optimus_net-4043.php.

[Liu et al., 2013] Liu, G., Tan, R., Zhou, R., Xing, G., Song, W.-Z., and Lees, J. M. (2013). Volcanic earthquake timing using wireless sensor networks. In *IPSN*.

[Liu et al., 2006] Liu, J., Wolfson, O., and Yin, H. (2006). Extracting semantic location from outdoor positioning systems. In *MDM*. Citeseer.

[Marvell Sheevaplug,] Marvell Sheevaplug. Marvell sheevaplug. www.plugcomputer.org.

[Meng et al., 2015] Meng, R., Shen, S., Roy Choudhury, R., and Nelakuditi, S. (2015). Matching physical sites with web sites for semantic localization. In *The 2nd workshop on Workshop on Physical Analytics*. ACM.

[Miluzzo, 2011] Miluzzo, E. (2011). *Smartphone Sensing*. PhD thesis, Dartmouth College.

[Moazzami et al., 2015] Moazzami, M., Phillips, D. E., Tan, R., and Xing, G. (2015). ORBIT: a smartphone-based platform for data-intensive embedded sensing applications. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks, IPSN 2015, Seattle, WA, USA, April 14-16, 2015*, pages 83–94.

[Moazzami et al., 2013] Moazzami, M.-M., Phillips, D. E., Tan, R., and Xing, G. (2013). A smartphone-based system platform for embedded sensing applications. Technical Report MSU-CSE-13-11, Dept. CSE, Michigan State University. http://www.cse.msu.edu/publications/tech/TR/MSU-CSE-13-11.pdf.

[Moazzami et al., 2017] Moazzami, M.-M., Singh, J., Srinivasan, V., and Xing, G. (2017). Deepcrowd-label: A deep-learning based crowd-assisted system for location labeling. In *4th International Workshop on Crowd Assisted Sensing, Pervasive Systems and Communications (CASPer 2017)*.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*.

[NASA PhoneSat 2013,] NASA PhoneSat 2013. Nasa phonesat project. http://open.nasa.gov/plan/phonesat/.

[Newton et al., 2009] Newton, R., Toledo, S., Girod, L., Balakrishnan, H., and Madden, S. (2009). Wishbone: Profile-based partitioning for sensornet applications. In *NSDI*.

[Object Tracking Robot,] Object Tracking Robot. Soccer robot project. https://code.google.com/p/android-object-tracking/.

[Oracle,] Oracle. Java annotations. http://docs.oracle.com/javase/tutorial/java/annotations/.

[Pan and Yang, 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*.

[Pei et al., 2013] Pei, L., Guinness, R., Chen, R., Liu, J., Kuusniemi, H., Chen, Y., Chen, L., and Kaistinen, J. (2013). Human behavior cognition using smartphone sensors. *Sensors*, 13(2):1402–1424.

[Pichotta and Mooney, 2016] Pichotta, K. and Mooney, R. J. (2016). Using sentence-level lstm language models for script inference. *arXiv preprint arXiv:1604.02993*.

[Ponnekanti et al., 2001] Ponnekanti, S. R., Lee, B., Fox, A., Hanrahan, P., and Winograd, T. (2001). Icrafter: A service framework for ubiquitous computing environments. In *Ubicomp 2001: Ubiquitous Computing*, pages 56–75. Springer.

[Quattoni and Torralba, 2009] Quattoni, A. and Torralba, A. (2009). Recognizing indoor scenes. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*.

[Ra et al., 2012] Ra, M.-R., Liu, B., La Porta, T. F., and Govindan, R. (2012). Medusa: a programming framework for crowd-sensing applications. In *MobiSys*.

[Raspberry Pi,] Raspberry Pi. Raspberry pi. http://www.raspberrypi.org.

[Rosen et al., 2004] Rosen, N., Sattar, R., Lindeman, R. W., Simha, R., and Narahari, B. (2004). Homeos: Context-aware home connectivity. In *International Conference on Wireless Networks*, pages 739–744.

[Sapiezynski et al., 2015] Sapiezynski, P., Stopczynski, A., Gatej, R., and Lehmann, S. (2015). Tracking human mobility using wifi signals. *PloS one*.

[Sharif Razavian et al., 2014] Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*.

[Shin et al., 2016] Shin, S., Hwang, K., and Sung, W. (2016). Quantized neural network design under weight capacity constraint. *arXiv preprint arXiv:1611.06342*.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[Sleeman and van Eck, 1999] Sleeman, R. and van Eck, T. (1999). Robust automatic p-phase picking: an on-line implementation in the analysis of broadband seismogram recordings. *Physics of the earth and planetary interiors*, 113.

[Snavely et al., 2006] Snavely, N., Seitz, S. M., and Szeliski, R. (2006). Photo tourism: Exploring photo collections in 3d. In *ACM SIGGRAPH*.

[Sorber et al., 2005] Sorber, J., Banerjee, N., Corner, M. D., and Rollins, S. (2005). Turducken: Hierarchical power management for mobile devices. In *MobiSys*.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[Taigman et al., 2014] Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708.

[también SCHUM,] también SCHUM, V. The evidential foundations of probabilistic reasoning.

[Tan et al., 2010] Tan, R., Xing, G., Chen, J., Song, W., and Huang, R. (2010). Quality-driven volcanic earthquake detection using wireless sensor networks. In *RTSS*.

[VolcanoSRI 2012,] VolcanoSRI 2012. Innovative monitoring systems help researchers look inside volcanos. www.cse.msu.edu/About/Notable.php?Nid=423.

[Wind et al., 2016] Wind, D. K., Sapiezynski, P., Furman, M. A., and Lehmann, S. (2016). Inferring stop-locations from wifi. *PloS one*.

[Yan et al., 2014] Yan, Y., Cosgrove, S., Anand, V., Kulkarni, A., Konduri, S. H., Ko, S. Y., and Ziarek, L. (2014). Real-time android with rtdroid. In *MobiSys*.

[Zamir et al., 2013] Zamir, A. R., Dehghan, A., and Shah, M. (2013). Visual business recognition: a multimodal approach. In *ACM Multimedia*. Citeseer.

[Zhang et al., 2015] Zhang, X., Yu, F. X., Chang, S.-F., and Wang, S. (2015). Deep transfer network: Unsupervised domain adaptation. *arXiv preprint arXiv:1503.00591*.

[Zhou et al., 2014] Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., and Oliva, A. (2014). Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*.

[Zilberstein, 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73.