This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

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

Mohammad-Mahdi Moazzami, *Student Member, IEEE,* Dennis E. Phillips, *Student Member, IEEE,* Rui Tan, *Member, IEEE,* and Guoliang Xing, *Member, IEEE*

Abstract—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. This paper presents ORBIT, a smartphone-based platform for data-intensive embedded sensing applications. ORBIT 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 that allows 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. 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.

Index Terms—Smartphone, embedded sensing, data processing, data-intensive applications.

1 Introduction

The ubiquity of smartphones and their multi-modal sensing capabilities have enabled a wide spectrum of mobile sensing applications. These applications are usually *human-centric* in that the smartphone utilizes on-board sensors to sense people and characteristics of their contexts. Different from these human-centric sensing applications, this paper considers an emerging class of smartphone-based *data-intensive embedded* sensing applications. In contrast to the people-centric nature of participatory sensing, smartphones in these applications are embedded into environments to sense and interact with the physical world autonomously over long periods of time. For instance, in the Floating Sensor Network project [1], 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 [2], volcanoes [3], and even operating miniature satellites [4]. Another important class of smartphone-based embedded systems is cloud robots [5], [6]. By integrating 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 embedded 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 [7].

However, several challenges must be addressed before smartphones can be used as a system platform for embedded sensing applications. These challenges include:

(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, smartphone operating systems (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

M. Moazzami, D. Philips, and G. Xing are with the Department of Computer Science, Michigan State University, 428 South Shaw Lane Room 3115 East Lansing, MI 48824, USA. E-mail: {moazzami, dennisp, glxing}@msu.edu.
R. Tan is with School of Computer Science and Engineering, Nanyang Technological University, N4-02C-117, 50 Nanyang Avenue, Singapore 639798. E-mail: tanrui@ntu.edu.sg.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

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 and 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.

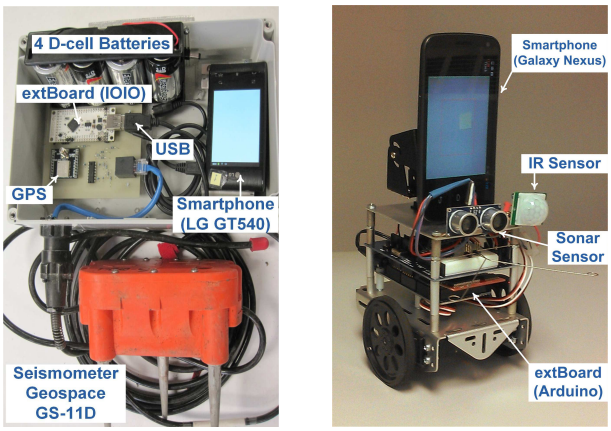
In this paper, we take a first step toward addressing these challenges collectively. We present ORBIT, a smartphone-based platform for embedded sensing systems. In particular, ORBIT leverages off-the-shelf smartphones to meet the energy-efficiency and timeliness requirements of data-intensive embedded sensing applications. ORBIT is based on a tiered architecture that consists up to three tiers: cloud, smartphone, and one or more energy-efficient peripheral boards (referred to as extBoard) that are interfaced with the smartphone.

This paper makes the following contributions. 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 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 extBoard. Several components of ORBIT bear some similarity with existing embedded system platforms [8], [9], [10], [11]. 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 three applications upon ORBIT: seismic sensing, visual tracking, 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 that ORBIT reduces energy consumption by up to 50% compared with baseline approaches.

2 System Overview

In this section, we present ORBIT, which is designed to address the three major challenges discussed in Section 1. An ORBIT node comprises an Android smartphone, an extBoard (e.g., IOIO [12] and Arduino [13]), and possibly a runtime system in the cloud. The extBoard is connected to the smartphone through a USB cable or bluetooth for communication. It is equipped with a low-power microcontroller (MCU), e.g., ATmega2560 with 16 MHz clock frequency, 8 KB RAM, and an analog-to-digital (A/D) convertor that can integrate various analog sensors. Fig. 1 shows two ORBIT prototypes, a seismic monitoring node and a robot sensing node used in the evaluation (cf. Section 5). Fig. 2 shows the overall system architecture.

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



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

Fig. 1. ORBIT nodes for seismic sensing and robots.

ORBIT is designed to meet the following three requirements. (1) Energy-efficiency while accounting for 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. ORBIT cannot achieve hard real-time guarantees. However, the statistical task timing model allows the task deadlines to be met with specified probabilities. (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 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. The major ORBIT components are described as follows.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

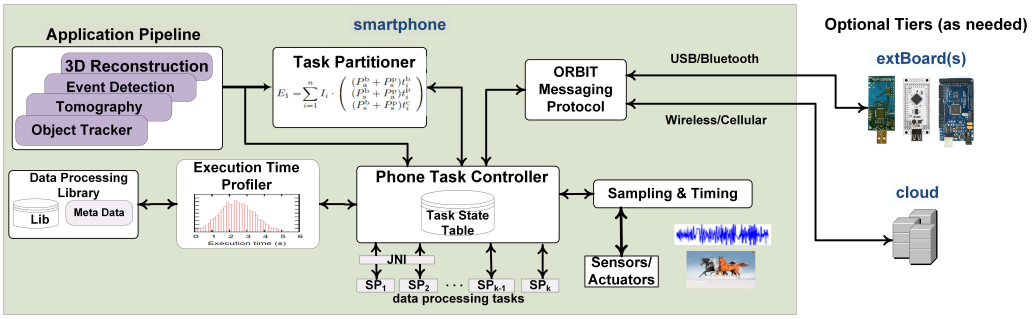


Fig. 2. System architecture of ORBIT. The shaded area indicates the components running on the smartphone tier.

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 this protocol are omitted in this paper and can be found in the previous work [14].

3 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. This 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, and adaptive delay-quality trade-off in the library are based on the findings of the measurement study in this section.

3.1 Timing Accuracy and Latency Profiling

Timing accuracy is critical for many sensing applications. For instance, acoustic or seismic source localization [15] 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,

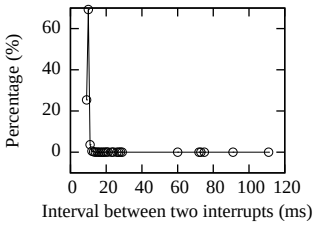


Fig. 3. Distribution of the intervals between two interrupts raised by a software timer of Android.

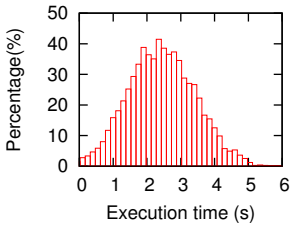


Fig. 4. Distribution of execution time of the SIFT algorithm on 640×480 images on Nexus S.

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.

Software Timer: Fig. 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.

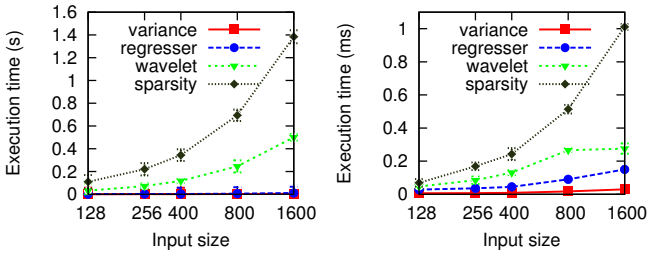
Event timestamping: We then measure the delay between the time instance when a pulse signal is received by a digital pin of an extBoard (which triggers a USB interrupt to Android) and when the USB interrupt is received in an Android application. Our measurement shows that this delay is highly variable 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. 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. 4 shows the distribution of the execution time of the scale-invariant feature transform (SIFT) used for detecting features of different 640×480 pixel images on a Nexus S. This example suggests that the statistical properties of the signal processing delays must be accounted for at run time 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing



(a) On Arduino. (b) On Nexus S.

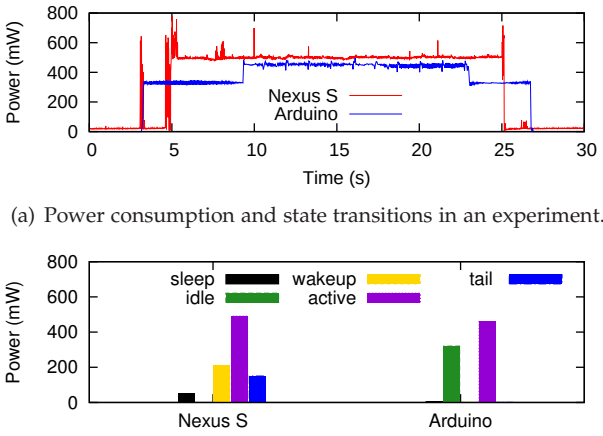
Fig. 5. Execution time of signal processing algorithms (error bar represents standard deviation).

the application. Fig. 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 3.2. Therefore, the smartphone can process the signals with less energy and shorter delays.

3.2 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. Specifically, we connect a precision resistor of 0.1 Ohm in series with the smartphone battery. The multimeter measures the voltage drop across the resistor and reports the measurements to a laptop computer via a USB cable. Fig. 6(a) 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 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. The two transitional phases at the 5th and 25th second are referred to as *wake-up* and *tail* phases, respectively. They last 200 ms and 755 ms approximately. There are also two spikes in Fig. 6(a), 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*. Fig. 6(b) shows the average power consumption under different states for the Nexus S and Arduino.

As we can also see in Fig. 6(a), the Arduino extBoard has three states, *active*, *idle*, and *sleep*. Its average current draws in these states are 90 mA, 66 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 and state transitions in an experiment.

(b) Power consumption comparison of different states.

Fig. 6. Nexus S and Arduino power consumption profiles.

3.3 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 [8], [10], we collectively report our measurement 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 executed by 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 energy consumption and latencies. When dispatching signal processing tasks to different tiers, these important characteristics must be considered to minimize the total system energy consumption while meeting application latency constraints.

4 Design And Implementation

4.1 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 a basic sensing or processing operation, such as computing mean and FFT. 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 3.1) and use it for task partitioning (as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

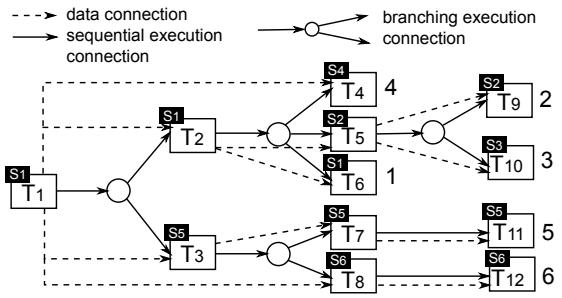


Fig. 7. An example ORBIT pipeline. The numbers besides the leaf nodes are the priorities assigned by the application developer; the tag S_x of a task represents the set it belongs to in the task partitioning solution.

```
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);
}
```

Listing 1. Pseudo-code for generating an application pipeline.

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 in Fig. 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 [16]. 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. 7 can simply be implemented as shown in Listing. 1, 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. If not all the outputs of one task is connected to the next task then the corresponding output is specified by the output ID. For example annotation $@Next\{T\_i\{k\}\}$ means the k-th output of the annotated task is connected to task i. The annotations $@source$ and $@sink$ are used to indicate the source and sink 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.

4.2 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 dimension reduction), to primitives like gradient decent optimization. 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. First, it is extensible so that developers can easily add more algorithms or port legacy signal processing libraries. Second, 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, 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.

4.2.1 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 optimal solution for an objective function experiences a longer execution time, consequently causing the application

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

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 [17]. This early-stopping mechanism for these iteration-based optimization tasks in ORBIT is controlled by three parameters: *stepSize, numOfIterations, 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.

4.2.2 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 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 will 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.

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 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, the kth thread 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.

ORBIT is designed to support a single and complex sensing application pipeline. The extension of ORBIT to support multiple applications competing for the resources such as processing cores is left for our future work.

4.3 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.

4.3.1 Power Management Model

From the key observations obtained from the measurement study in Section 3, 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. 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 cloud messages. Fig. 8 illustrates the extBoard's duty cycle and the smartphone on-demand sleep schedule.

4.3.2 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

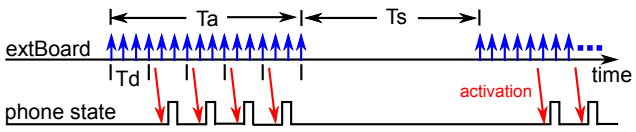


Fig. 8. Power management scheme.

and be subject to change after deployment, as shown in the Fig. 4 example. To address this issue, ORBIT continuously measures the latency of each task at runtime and periodically updates 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 the 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}, for each task i on the smartphone, extBoard and cloud respectively (cf. Section 4.3.3). Under this approach, ORBIT can achieve optimal partitioning solution while meeting the timing requirements statistically.

4.3.3 Partitioning with Sequential Execution

As discussed in Section 4.3.1, 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.

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 4.4.2), it is safe to set D_b to a value that is slightly smaller than the sampling period. Existing studies [8], [10] adopt an integer linear program (ILP) which minimizes a linear combination of the network bandwidth and the CPU consumption subject to the upper bounds for these resources. It is important to note that under the conventional ILP partitioning, the model only accounts for the execution time latency (i.e., CPU consumption) and data copy latency between tiers (e.g., network bandwidth). 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 also considers 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. 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, 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. We have

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

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

where E_w, T_w, E_T and T_T are the energy consumed and the time spent during the tail (T) and wake-up (w) phases, respectively.[2] 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 $I_{i-1} = (0,1,0)$, $d(I_i, I_{i-1}) = 1$. The maximum distance is $d_{max}(I_i, I_{i-1}) = 2$ when a task is previously assigned to the extBoard (i.e., $I_{i-1} = (1,0,0)$) and then to the cloud (i.e., $I_i = (0,0,1)$), or vice-versa. In this case, the data has to be transferred between these two tiers through smartphone because there is no direct communication between extBoard and the cloud server. Moreover, the execution times of the extBoard, smartphone and cloud portion of the application are given by:

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

2. E_T and E_w are the additive energy consumption of the entire device, rather than a single component.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

(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 paramter of T_i needs to be copied between the smartphone and the cloud or between the smartphone and the extBoard, 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,

$$E_2 = \sum_{i=1}^{n}\sum_{j=1}^{J_i} d(I_i, I_{s(i,j)}) P_{\mathrm{a}}^{\mathrm{p}} t_{\mathrm{c}} l_{i,j},\ \ \tau_2 = \sum_{i=1}^{n}\sum_{j=1}^{J_i} d(I_i, I_{s(i,j)}) t_{\mathrm{c}} l_{i,j}.$$

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 formulas of E and τ. ORBIT uses brute-force search to find the optimal solution. As the number of tasks in an application is often small, our measurements show that the brute-force search introduces little overhead even if the Task Partitioner is executed periodically by the smartphone.

4.3.4 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. 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. 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. 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 4.3.3 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. 7, if it decides to execute T_5 according to the result of T_2, it will dispatch T_2 according to S_2.

We now discuss an approach to learn the branch priorities at run time. Specifically, the system starts with an initial priority assignment (e.g., initialized to equal values or specified by the developer) and then measures the execution frequency of each branch at run time. Based on the measured frequencies, the system updates the branches' priorities accordingly. Once the branch with the maximum priority changes, the system re-partitions the pipeline. Alternatively, ORBIT can re-partition the pipeline when any branch priority or a certain number of branches priorities have changed. However, this approach yields more frequent partitioning that leads to more system overhead. For instance, each change in the task partition requires the system to preserve the application state and move the tasks' snapshots between different tiers. Therefore, the application developer may apply a policy on when to trigger the re-partition to achieve a satisfactory trade-off between the benefit of being more adaptive to the priority changes and the overhead caused by updating the task partition.

Based on the application pipeline in Fig. 7, we conduct a simulation to evaluate and illustrate the above adaptive mechanism. Fig. 9 shows the simulation results after multiple executions of the application pipeline. In this experiment each task has a predefined delay (based on the measurement study in Section 3). To simulate the runtime branching behavior, the application randomly branches to a next task. Therefore, the simulation builds a distribution of the execution of each branch over the time. The branches are ranked based on their frequency of execution within a time window. Branches with higher frequencies of execution (illustrated by thicker edges in Fig. 9) are assigned with higher priorities. As we can see, the branch priorities are updated over the time. In iterations 4, 7 and 15 (Figs. 9(c), 9(f), 9(j)), the branch with the highest priority changes and ORBIT re-partitions the pipeline.

4.4 Task Controllers

Task Controllers (TCs) on the smartphone, the extBoard, and the cloud execute the sensing application according to the assignment computed by the Task Partitioner. Fig. 2 shows the interaction of the TCs with other ORBIT components.

4.4.1 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

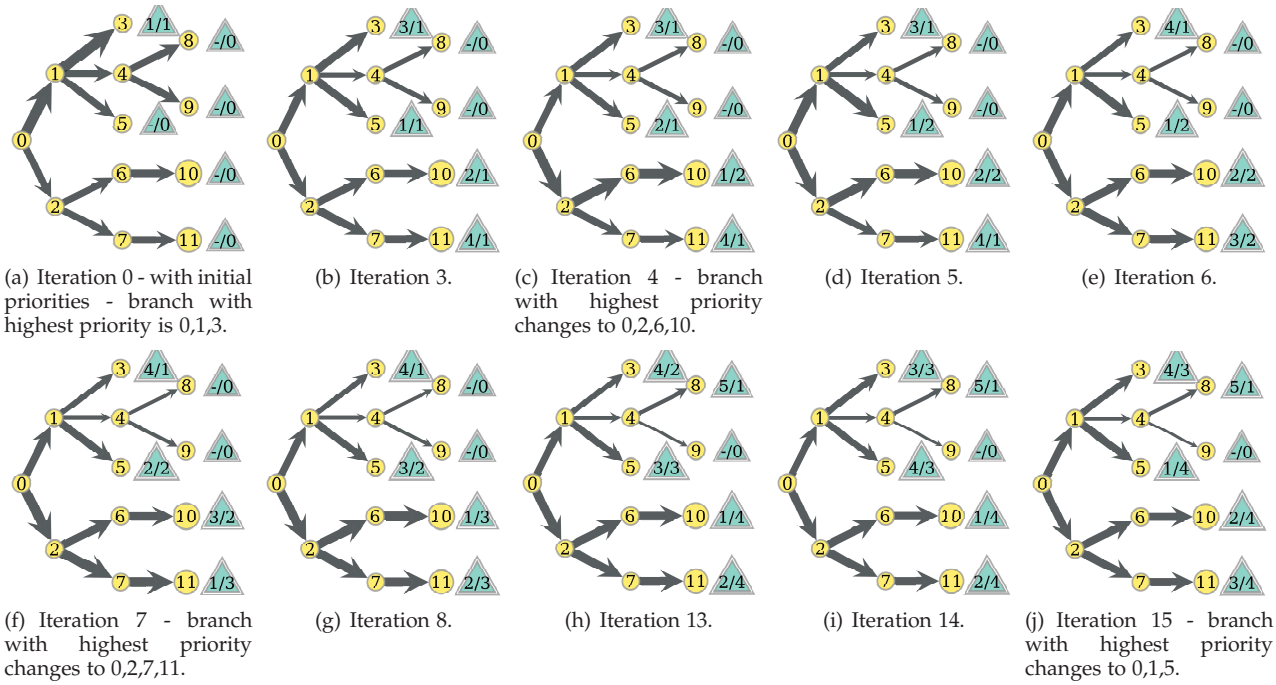


Fig. 9. Learning branch priorities over the time and adapting the partitioning accordingly. The numbers "x/y" besides a leaf node: "x" is the learned priority, "y" is the measured execution counter. If a branch is not executed, its priority is not defined (i.e., "−") and its execution counter is 0. An iteration means an execution cycle of the application pipeline.

the first task. When the smartphone is not executing a task, it switches to the sleep state to conserve energy. When the task execution will return to the smartphone, a message is sent to wake up the phone and activate its TC to execute next tasks. The phone TC continuously updates the task meta information (e.g., execution times) and branch priorities.

Our measurement study shows that the smartphone consumes considerable energy during wake-up and tail phases (cf. Section 3.2). We optimize the design of TC to start a task once the smartphone wakes up or to let the smartphone sleep once 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 [14]. 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. But typical signal processing pipelines likely contain a limited number of branches.

4.4.2 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 daemon that checks for the arrival messages from one or multiple smartphones. There are two types of tasks in the cloud: the compute-intensive tasks 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 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.

5 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 use different applications. 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% to 15%. The memory usage is about 22.5 MB to 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

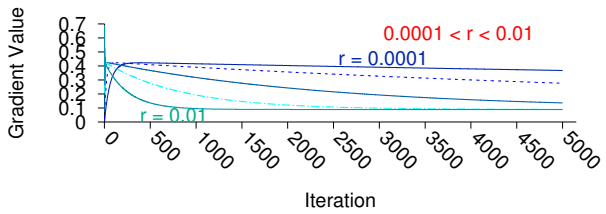
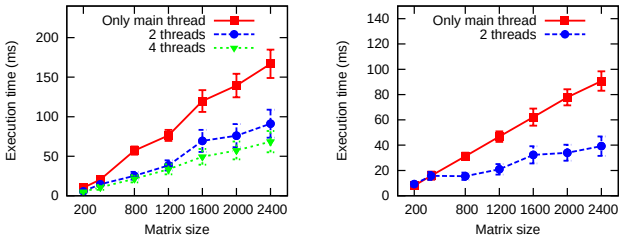


Fig. 10. Delay-quality trade-off (r denotes step size).



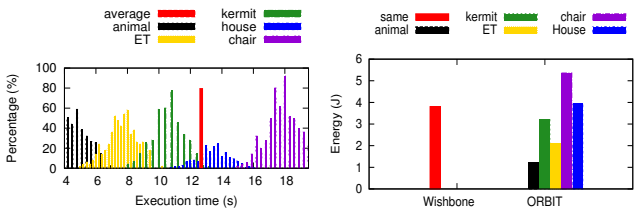
(a) Moto-G (4 cores). (b) Galaxy Nexus (2 cores).

Fig. 11. Smartphone multi-threading reduces processing delay of compute-intensive tasks.

Delay-quality trade-off: In section 4.2.1, we discussed how algorithms are tuned for desirable trade-offs between quality and delay. Fig. 10 shows the convergence of the gradient descent algorithm for different step sizes r and number of iterations. As expected, we can see that the gradient value decreases with the number of iterations, until finally converges to the solution. Larger step sizes result in faster convergence speeds. However, the rate of decrease slows after a certain iteration for each step size, meaning that the task does not benefit much 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 losing a significant accuracy. Examples of algorithms that can benefit from this feature include SVM, linear regression and K-mean clustering. This feature not only provides insights for 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 4.2.2, smartphone TC can partition compute-intensive tasks into multiple threads to reduce the processing delays. Fig. 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. The 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 two threads. It also reduces by 56.1% for Mote-G when the computation is partitioned into four threads.

We can also see that 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 four threads in Moto-G do not provide significant improvement over two threads. This is because, once the computation is partitioned into two threads, the problem size is reduced by half. Consequently, when each thread is further split into two 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.



(a) Execution time distribution of SIFT. (b) Energy consumption of SIFT for different input images.

Fig. 12. The data-dependent algorithms.

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 in Section 3 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. 12(a) shows the distribution of the execution time for the SIFT algorithm for input images with different dimensions and number of SIFT features. Fig. 12(b) 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 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.

6 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. These applications are summarized in Table 1. Each application demonstrates different facets of ORBIT, e.g., different numbers of tasks

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

TABLE 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

in the pipeline, the use of different sensors, different tiers, and different data fidelity requirements. Our goal is to demonstrate the capabilities and effectiveness of the platform rather than present novel applications.

6.1 Robotic Sensing

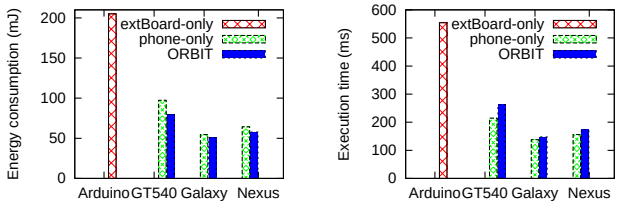
We choose a robotic sensing application for our first case study. In this case study, the application estimates the presence, the distance, and the direction of an object approaching using an infrared (IR) and a sonar sensor attached to the robot as well as the smartphone's built-in camera (cf. Fig. 1(b)). Once an object is detected by IR sensor, its distance is estimated using the sonar sensor. If it is within a specified range, the extBoard sends a command to the smartphone to wake it up and activate the camera.

Once the system captures the image, it converts the image to grayscale and computes the threshold to separate the background and the foreground. Then, 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 [18].

In this case study, a few tasks are not allowed to be partitioned since they directly sample the sensors or actuate the 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 it with two baselines. The baselines run all tasks on the extBoard only (extBoard-only) or the smartphone only (smartphone-only). Fig. 13 shows the energy consumption and the application execution time when the total processing delay bound (D) is set to 0.4 s. Fig. 13(a) and Fig. 13(b) plot the estimated total energy consumption and total execution time (i.e., smartphone + extBoard) of an 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 and smartphone energy consumption. Assume that at least n frames are



(a) Total energy consumption. (b) Total execution time.

Fig. 13. The results of various partition schemes. Nexus refers to Nexus S and Galaxy refers to Galaxy Nexus.

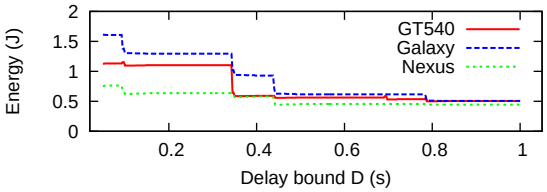


Fig. 14. Impact of delay bound setting on the task assignment and total energy consumption. Nexus refers to Nexus S and Galaxy refers to Galaxy Nexus.

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 denote the speed of the object. The time that the object 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 that the delay bound is inversely proportional to the object's velocity. For example, if the camera's angle of view is $18°$, 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 process each frame within 0.28 seconds.

Fig. 14 shows the smartphone energy consumption versus D. We can see that the total energy consumption decreases with D. This is because, with a higher delay bound, more tasks are assigned to the extBoard, allowing the smartphone to sleep for a longer time period. Note that the extBoard keeps active to continuously sample the sensors and control the robots servos. Thus, as the result suggests, it is more energy efficient to execute more tasks on the extBoard as long as the application's timing requirement, D, is met. The smartphone only wakes up when an image must be captured. It then preprocesses the image and sends the result to the extBoard to detect the object.

6.2 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 [15] 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. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

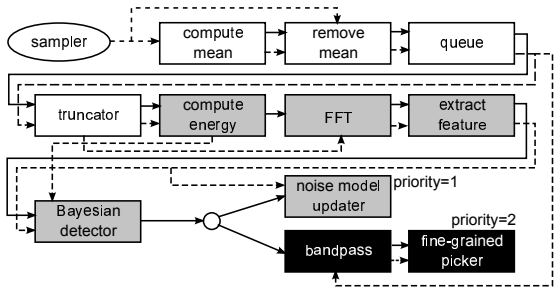


Fig. 15. The block diagram of the seismic event timing application.

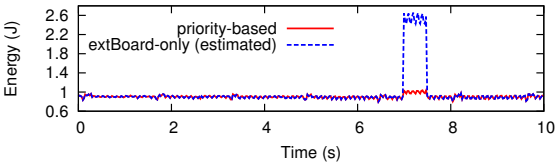
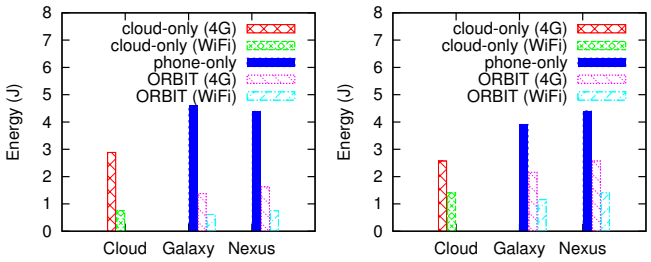


Fig. 16. The energy consumption trace of a node.

context of earthquake detection, the system must have a response time of several seconds. In this section, we focus on evaluating the effect of branches based on this case study.

To evaluate the effect of branches, we integrate the event timing application in [14] with an event detection approach [19]. Fig. 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 [19], 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 [20] 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 the most frequently. In this case, all tasks except the compute-intensive *bandpass* and *fine-grained picker* are assigned to the extBoard. Fig. 16 plots the trace of energy consumption of an 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 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. The *extBoard-only* approach cannot meet the delay bound when the detector makes positive decisions. Thus, we cannot directly run this approach on the node. Instead, we estimate the energy consumption based on the extBoard power model and task meta records. Fig. 16 shows that, when an event occurs, the *extBoard-only* consumes higher power than our approach, due to the long execution time of *fine-grained picker* on the extBoard.



(a) House image (1280 × 722px). (b) kermit image (640 × 480px).

Fig. 17. The results of various partition schemes.

6.3 Multi-Camera 3D Reconstruction

This case study is inspired by Phototourism [21] and involves opportunistic sensing wherein smartphone-equipped robots capture location-based images to collaboratively reconstruct a 3D structure. Compared with the previous two case studies, this application is partitioned cross 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. 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. 17(a) and Fig. 17(b), 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. 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 yields the most energy efficient partition. In addition, this result demonstrates that ORBIT considers both the impacts of the input data size and content on the execution time of data processing tasks. Existing task partitioning approaches [8], [10] often fall short of addressing these two affecting factors explicitly.

6.4 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 with baseline approaches.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2016.2573825, IEEE Transactions on Mobile Computing

An interesting observation from case studies 1 and 2 is the system power consumption is highly probabilistic. For instance, whether the energy-consuming image processing is needed in case study 1 depends on the decision of the first stage IR/sonar sensing which is subjected 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. As a possible improvement, ORBIT could be provided runtime feedback such as the detection history and estimated system detection performance. This would allow ORBIT to optimize the wiring of tasks and priorities of tasks to reduce power consumption.

Case study 3 suggests that the input data affects the partitioning result. For example, when the input data is simple, 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. The runtime decision making shows ORBIT's intelligent flexibility.

7 Related Work

Various task offloading schemes for smartphones have been developed recently. Spectra [22] allows programmers to specify task partitioning plans given application-specific service requirements. Chroma [23] aims to reduce the burden on manually defining the detailed partitioning plans. Medusa [24] features a distributed runtime system to coordinate the execution of tasks between smartphones and cloud. Turducken [11] adopts a hierarchical power management architecture, in which a laptop can offload lightweight tasks to tethered personal digital assistants 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 save energy.

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 and 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. More details about this messaging protocol is described in the previouse work [14].

The MAUI system [8] 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 [10] also features a task dispatch scheme. 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 significant variations in real scenarios. To address this, ORBIT measures the statistical timing profiles at run time, and periodically refines the partitioning results. Moreover, Wishbone formulates the partitioning problem as a $0/1$ integer 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 [25] tackles the lack of hard real-time capability of Android system 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 can run on RTDroid and the ORBIT-based sensing applications can benefit from both.

Similar to ORBIT, EmStar [9] 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 data-intensive embedded sensing that they are not originally designed for.

8 Conclusion

This paper presents ORBIT, a smartphone-based platform for data-intensive and 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.

Moreover, we plan to extend the platform and the power model to address multiple simultaneous applications. Optimizing the total energy consumption of the system with multiple application pipelines running on the same tiered architecture is challenging and requires further extensions to ORBIT and the power model, where ORBIT needs to act as an operating system to deal with the applications' contention of resources. One possible solution is to combine multiple pipelines as a single pipeline, and then apply our current partitioning algorithm that addresses execution branches. The automatic combination will be an interesting and challenging problem.

Acknowledgments

This work was supported in part by U.S. National Science Foundation under grants CNS-1218475, OIA-1125163, and

CNS-0954039 (CAREER), and in part by a Start-up Grant at Nanyang Technological University.

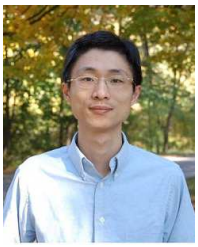
REFERENCES

[1] "Floating sensor network," http://float.berkeley.edu.
[2] M. Faulkner, M. Olson, R. Chandy, J. Krause, K. M. Chandy, and A. Krause, "The next big one: Detecting earthquakes and other rare events from community-based sensors," in *The 10th International Processing in Sensor Networks (IPSN)*, 2011.
[3] "Innovative monitoring systems help researchers look inside volcanos," www.cse.msu.edu/About/Notable.php?Nid=423.
[4] "Nasa phonesat project," http://open.nasa.gov/plan/phonesat/.
[5] E. Guizzo, "Robots with their heads in the clouds," *IEEE Spectrum*, vol. 48, no. 3, pp. 16–18, 2011.
[6] "Cloud robotics," http://goldberg.berkeley.edu/cloud-robotics/.
[7] "LG Optimus Net," http://www.gsmarena.com/lg_optimus_net-4043.php.
[8] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *The 8th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.
[9] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "Emstar: A software environment for developing and deploying wireless sensor networks," in *USENIX Annual Technical Conference*, 2004.
[10] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based partitioning for sensornet applications," in *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
[11] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins, "Turducken: Hierarchical power management for mobile devices," in *The 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2005.
[12] "IOIO for Android," www.sparkfun.com.
[13] "Arduino board," http://www.arduino.cc.
[14] M.-M. Moazzami, D. E. Philips, R. Tan, and G. Xing, "ORBIT: A smartphone-based platform for data-intensive embedded sensing applications," in *The 14th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.
[15] G. Liu, R. Tan, R. Zhou, G. Xing, W.-Z. Song, and J. M. Lees, "Volcanic earthquake timing using wireless sensor networks," in *The 12th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.
[16] Oracle, "Java annotations," http://docs.oracle.com/javase/tutorial/java/annotations/.
[17] S. Zilberstein, "Using anytime algorithms in intelligent systems," *AI magazine*, vol. 17, no. 3, p. 73, 1996.
[18] "Soccer robot project," https://code.google.com/p/android-object-tracking/.
[19] R. Tan, G. Xing, J. Chen, W. Song, and R. Huang, "Quality-driven volcanic earthquake detection using wireless sensor networks," in *The 31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.
[20] R. Sleeman and T. van Eck, "Robust automatic p-phase picking: an on-line implementation in the analysis of broadband seismogram recordings," *Physics of the earth and planetary interiors*, vol. 113, 1999.
[21] N. Snavely, S. M. Seitz, and R. Szeliski, "Photo tourism: Exploring photo collections in 3D," in *The 33rd International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2006.
[22] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *The 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.
[23] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *The 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.
[24] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan, "Medusa: a programming framework for crowd-sensing applications," in *The 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.
[25] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, and L. Ziarek, "Real-time android with rtdroid," in *The 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

Mohammad-Mahdi Moazzami is a Ph.D. candidate in the Department of Computer Science and Engineering at Michigan State University (MSU). He received the B.S. from the Computer Engineering Department at Sharif University of Technology, Tehran, Iran, in 2007, and M.S. degree in computer science from MSU in 2011. He worked in industry for couple of years before and during the grad school. His research interests include smartphone-based sensing applications, scalable machine learning algorithms, and distributed information processing and mobile computing systems. His paper in 2013 IEEE Intl. Conf. Pervasive Computing & Communications (PerCom) was the Best Paper Award Runner-Up, and his smartphone app was the winner of the silver prize in the 2015 ACM Intl. Conf. on Mobile Computing & Networking (MobiCom) Mobile Apps Contest.

Dennis E. Phillips is a Ph.D. candidate in the Department of Computer Science and Engineering at Michigan State University (MSU). He received the B.S. (1975) degree in computer science with minor in electrical engineering and M.S. (1977) degree in computer science from MSU, and M.A. (1990) degree in education from Tufts University, Somerville, MA. He worked 14 years in industry in the areas of distributed systems. Prior to his Ph.D. study, he supported the U.S. Department of Defense in the areas of network security and general IT support services. His research interests include wireless sensor networks, machine learning and distributed information processing. His paper in 2013 IEEE Intl. Conf. Pervasive Computing & Communications (PerCom) was Best Paper Award Runner-Up.

Rui Tan is an Assistant Professor at School of Computer Science and Engineering, Nanyang Technological University, Singapore. Previously, he was a Research Scientist (2012–2015) and a Senior Research Scientist (2015) at Advanced Digital Sciences Center, a Singapore-based research center of University of Illinois at Urbana-Champaign (UIUC), a Principle Research Affiliate (2012–2015) at Coordinated Science Lab of UIUC, and a postdoctoral Research Associate (2010–2012) at Michigan State University. He received the Ph.D. (2010) degree in computer science from City University of Hong Kong, the B.S. (2004) and M.S. (2007) degrees from Shanghai Jiao Tong University. His research interests include cyber-physical systems, sensor networks, and pervasive computing systems. His papers in 2013 IEEE Intl. Conf. Pervasive Computing & Communications (PerCom) and 2014 ACM/IEEE Intl. Conf. Information Processing in Sensor Networks (IPSN) were Best Paper finalists.



Guoliang Xing received the B.S. degree in electrical engineering from Xi'an Jiao Tong University, China, in 1998, and the M.S. and D.Sc. degrees in computer science and engineering from Washington University in St. Louis, in 2003 and 2006, respectively. He is an Associate Professor in the Department of Computer Science and Engineering at Michigan State University. He received the NSF CAREER Award in 2010. He received the Best Paper Awards at 2010 IEEE Intl. Conf. Network Protocols (ICNP) and 2012 ACM/IEEE Conf. Information Processing in Sensor Networks (IPSN) SPOTS track. His research interests include cyber-physical systems for sustainability, smartphone systems, and wireless networking.