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

Mohammad-Mahdi Moazzami[*] Dennis E. Phillips[*] Rui Tan[†] Guoliang Xing[*]
Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA
Advanced Digital Sciences Center, Illinois at Singapore

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 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. Extensive microbenchmark evaluation and two case studies including seismic sensing and multi-camera 3D reconstruction, validate the generic design of ORBIT.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems, signal processing systems

Keywords

Smartphone, embedded sensing, data processing, data-intensive applications

IPSN'15, April 14-16, 2015, Seattle, WA, USA
Copyright 2015 ACM 978-1-4503-3475-4/15/04$15.00
http://dx.doi.org/10.1145/2737095.2737098.

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 [9], 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 [7], volcanoes [13], and even operating miniature satellites [22]. Another important class of smartphone-based embedded systems is cloud robots [11] [4]. 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 [18].

However, several challenges must be addressed before smartphones can be used as a system platform for embedded sensing applications. First, the smartphones, which are designed to provide several days of battery life, are ill-suited for many embedded sensing applications that must operate unattended for long periods of time. Many of today's embedded applications are inherently *data-intensive* in that sensors must sample at high rates (e.g., 100 Hz in seismic sensing [29]). The continuous sensor sampling can prevent the smartphone from entering sleep state, leading to battery depletion in a few hours. Moreover, the current major smartphone operating systems (OSes) do not provide real-time functionalities, such as constant sampling rate, precise timestamping, and programming interfaces for expressing timing requirements, which are crucial to many embedded sensing applications. For instance, our measurements show that the USB hardware interrupt of Android phones suffers an unpredictable delay of up to 5 ms, which makes it impossible to achieve a high constant sampling rate. Lastly, the smartphone programming environment, although simplifying many programming tasks in the life cycle of embedded systems such as debugging, remote data logging, visualization, and software maintenance, lacks important embedded programming support such as resource-efficient signal processing libraries and communication/-control primitives for peripheral sensors.

In this paper, 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 [1] and IOIO [14]. 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.

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 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 [5, 10, 23, 28]. 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. 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 [16, 17, 15] and sensing algorithms such as context classifiers [3]. Recently, smartphones have been used in a number of embedded sensing applications. In [7], smartphones are used to build an earthquake early warning system using an onboard accelerometer. In the Floating Sensor Network project [9], 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 [22] has launched low-cost satellites equipped with Android smartphones. Controlled by a smartphone, such small satellites could perform various tasks such as earth observation and space debris tracking. Several recent efforts focus on building *cloud robots* [11] 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 [8] allows programmers to specify task partitioning plans given application-specific service requirements. Chroma [2] aims to reduce the burden on manually defining the detailed partitioning plans. Medusa [25] features a distributed runtime system to coordinate the execution of tasks between smartphones and cloud. Turducken [28] 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 smartphone-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 MAUI system [5] 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.

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

The Wishbone system [23] 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 nonlinear optimization problem and supports three or more tiers.

RTDroid [30] 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 [10] 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.

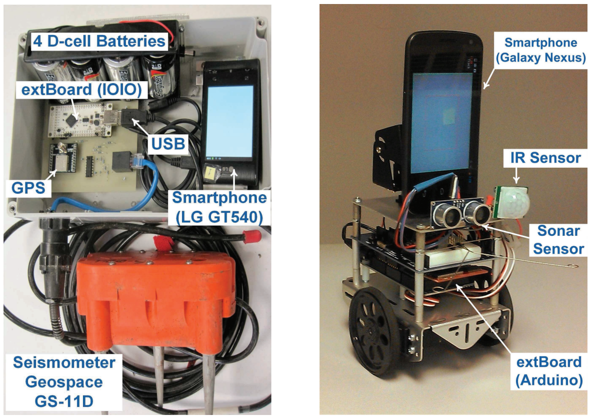
3. MOTIVATION AND SYSTEM OVERVIEW

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.

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 [12], SheevaPlug [20], and Raspberry Pi [26], 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).



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

Figure 1: ORBIT nodes for seismic sensing and robots.

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, 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.

3.2 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 [14] and Arduino [1]), 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. 1 shows two ORBIT prototypes, a seismic monitoring node and a robot sensing node that are used in the evaluation (cf. Section

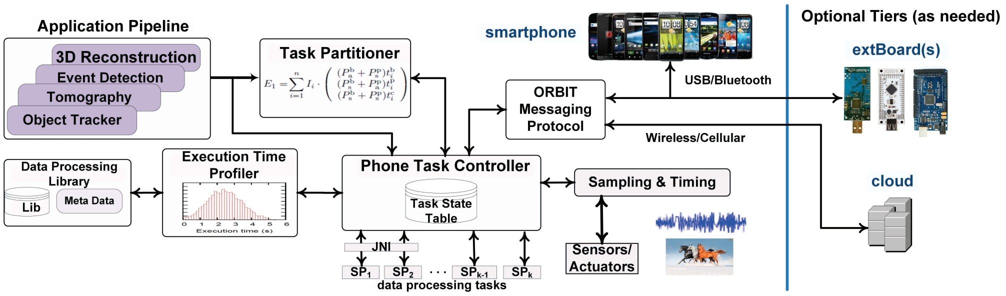


Figure 2: System Architecture of ORBIT.

6). Fig. 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).

4. 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 or ORBIT messaging protocol are based on the findings of the measurement study discussed in this section.

4.1 Timing Accuracy and Latency Profiling

Timing accuracy is critical for many sensing applications. For instance, acoustic or seismic source localization [19] 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.

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 sam-

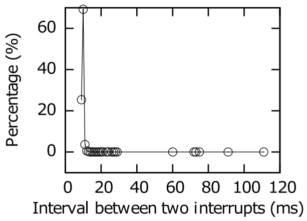


Figure 3: Distribution of the intervals between two interrupts raised by a software timer of Android.

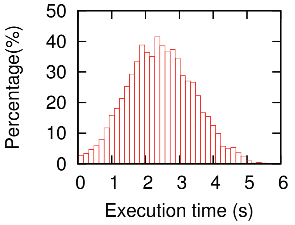
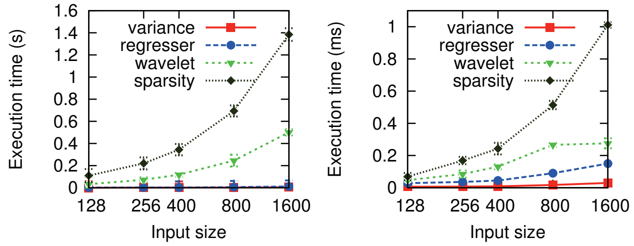


Figure 4: Distribution of execution time of the SIFT algorithm on 640x480 images on Nexus S.



(a) On Arduino

(b) On Nexus S

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

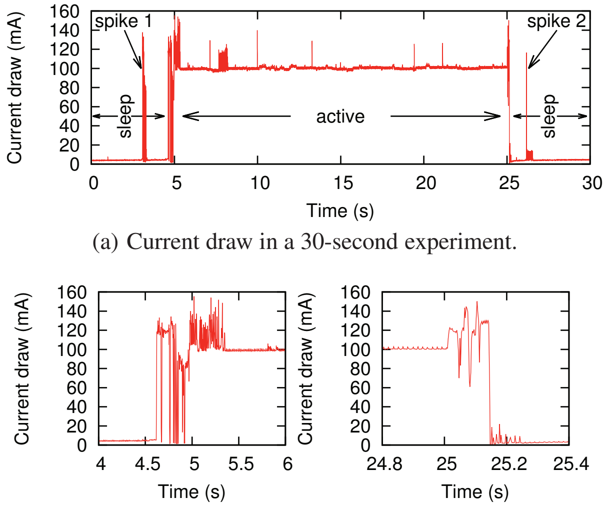
pling or precise event timestamping. In contrast, our measurement shows that the timing error of an Arduino extBoard is no greater than 12 μs, due to the availability of hardware timers and efficient interrupt handling.

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. 4 shows the distribution of the execution 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. 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 4.2. Therefore, the smartphone can process the signals with less energy and shorter delays.

4.2 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. 6(a) shows the current draw of a Samsung Nexus S 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. Fig. 6(b) 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. 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*.



(a) Current draw in a 30-second experiment.

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

Figure 6: Nexus S current draw profiles.

The Arduino extBoard has three states, *active*, *idle*, and *sleep*. Its average current draw 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.

4.3 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 [23, 5], 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 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.

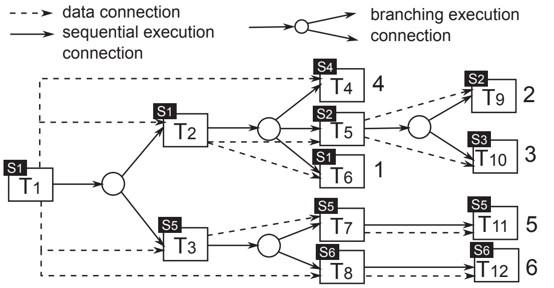


Figure 7: An example ORBIT application. (The numbers besides the leaf nodes in the execution tree are the priorities assigned by the application developer; the tag S_x of a task represents the set it belongs to in the task partitioning solution.)

5. DESIGN AND IMPLEMENTATION

This section presents the design of ORBIT to achieve the objectives discussed in Section 3.1.

5.1 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) → *low pass filter* → *face recognition* → *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 4.1) 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 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 [24]. 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. The annotations $@source$ and $@sink$ are used to indicate the source and sink tasks in the pipeline.

Listing 1: pseudo-code for generating an application pipeline

```
/** 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);
}
```

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.

5.2 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 processing 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.

5.2.1 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 [31]. This *early-stopping* mechanism for these iterative-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.

5.2.2 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., $\mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_K]$, 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 $\mathbf{y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k]$. 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 $\mathbf{y} = [\mathbf{y_1}, \mathbf{y_2}, \dots \mathbf{y_k}]$ 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.

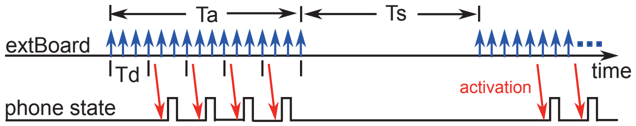


Figure 8: Power management scheme.

5.3 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.

5.3.1 Power Management Model

From the key observations obtained from the measurement-based study in Section 4, 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.

5.3.2 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. 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.

5.3.3 Partitioning with Sequential Execution

As discussed in Section 5.3.1, 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: $\mathbb{T} = T_1 \to T_2 \to \ldots \to T_n$. 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 [21].

Task Partitioning Problem. *For the sequential execution $\mathbb{T} = T_1 \to T_2 \to \ldots \to T_n$, 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 it is shown in [23] and [6] 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.

5.4 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 shows the interaction of the TCs with other components in ORBIT.

5.4.1 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 4.2). 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 (cf. section 5.5). 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.

5.4.2 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.

5.5 ORBIT Messaging Protocol

ORBIT supports sensing applications connecting to a variety of external sensors over wired and wireless channels as well as many built-in smartphone sensors. However, ensuring proper device interaction can be burdensome, especially when a single application needs to integrate a number of sensors using different communication channels and data formats. This is even more tedious when the application has to transfer the data between different tiers in a multi-tier architecture. This section presents ORBIT Messaging Protocol (OMP) which simplifies data transfer with abstractions separating responsibilities between the extBoard, the smartphone, and the cloud platform.

Smartphones communicate with peripherals through two USB modes (USB accessory and USB host) and Bluetooth. Android Bluetooth and USB communications only provide a byte stream communication channel. This channel neither provides delivery guarantees, nor supports the concept of messages. On the other hand, as it is discussed in Section 5.1, the communication unit between different tasks in a sensing application formed as a task structure is a *message*. To achieve this, ORBIT Messaging Protocol implements simple message framing along with message checksums

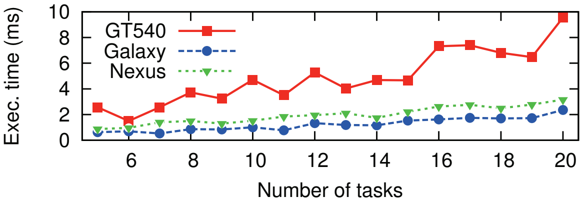


Figure 9: Execution time of Task Partitioner.

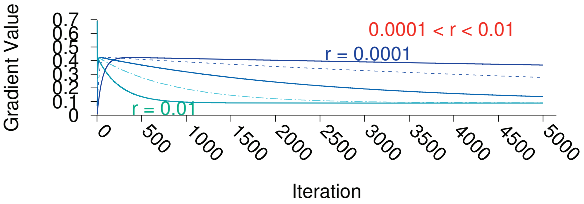


Figure 10: Delay/quality trade-off (r = step size)

to provide a more robust communication channel. Also, ORBIT provides message segmenting and queuing mechanism for long messages. In order to have homogeneity between communication with tiers, ORBIT implements the same messaging protocol for the cloud communication on top of the HTTP protocol. This facilitates a modular framework that allows developers to focus on writing minimal pieces of data transfer and sensor specific code.

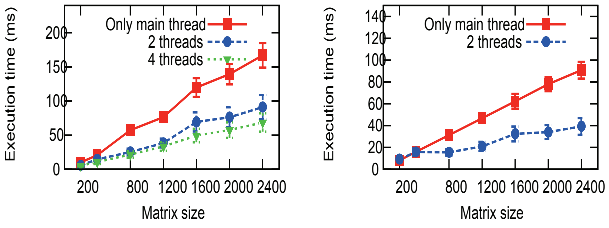
6. MICROBENCHMARK

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.

Overhead of online partitioning: As discussed in Section 5.3, the Task Partitioner runs online to adapt to the variable execution time of tasks. Fig. 9 shows the execution delay of the Task Partitioner on various smartphones versus the number of tasks. We can see that the Task Partitioner takes less than 10 milliseconds when there are 20 tasks. In addition, the Task Partitioner is called only when there is substantial change of task execution time. Therefore, the online task partitioning does not introduce significant runtime overhead on the smartphone.

Delay/quality trade-off: In section 5.2.1 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 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 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.



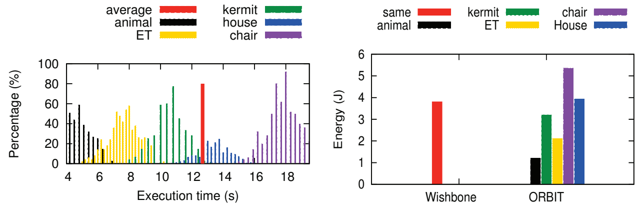
(a) Moto-G (4 cores) (b) Galaxy Nexus (2 cores)

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

Effect of data partitioning and multi-threading: As discussed in Section 5.2.2, 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, $\mathbf{y} = \mathbf{A}\mathbf{x}$, 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 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. 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



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

Figure 12: The data-dependant algorithms.

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.

7. CASE STUDIES

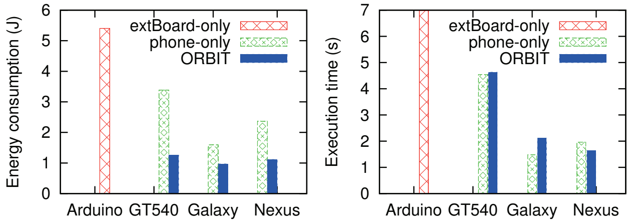
To demonstrate the expressivity of ORBIT application scripting as well as the generality and flexibility of ORBIT as a platform, we have prototyped three different embedded sensing yet data-intensive applications (cf. Table 1). Each application demonstrates different facets of ORBIT varying the number 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. Due to space limitations only the Event Timing and Multi Camera 3D Reconstruction case studies are presented here. The Robotic Sensing case study can be found a technical report [21].

Table 1: ORBIT based applications.

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

7.1 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 [19] 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 preprocessed 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.



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

Figure 13: The results of various partition schemes.

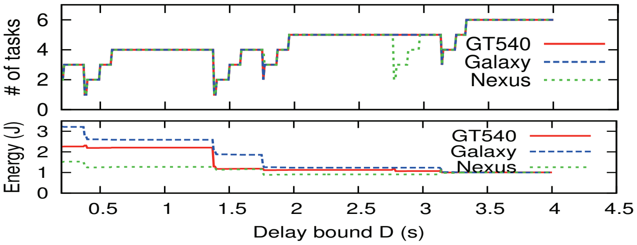
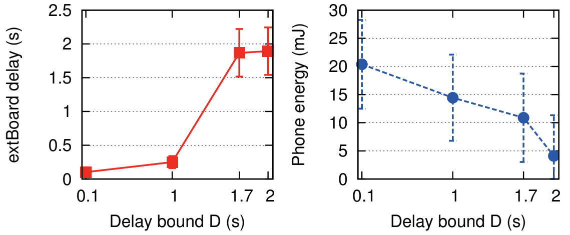


Figure 14: Impact of delay bound setting on the task assignment and total energy consumption. Top: The number of tasks assigned to the extBoard versus delay bound. Bottom: Total energy consumption versus delay bound.

Effectiveness of Task Partitioner: We first evaluate the effectiveness of the task partitioning algorithm presented in Section 5.3.3, by comparing the following partitioning approaches: extBoard only, phone-only, and greedy. The greedy approach assigns as many processing tasks to the extBoard as can be supported by the delay bound. Fig. 13 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. 13(a) and Fig. 13(b) 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. 14 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 5.3.3. However, we can see a number of drops in the top portion of Fig. 14. 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 reduce the smartphone energy consumption. The bottom portion of Fig. 14 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. 15 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. 15(a) we can see that the specified delay bound is al-



(a) extBoard processing delay (b) Phone energy consumption

Figure 15: The measured extBoard processing delay and smartphone energy consumption versus delay bound.

ways met. Moreover, the extBoard processing delay increases with the delay bound, proving the effective utilization of the allowed extBoard CPU time. From Fig. 15(b), the smartphone energy consumption decreases with the delay bound, which is consistent with our analysis.

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. 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. 15(b), 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 4), 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.

7.2 Multi-camera 3D reconstruction

The final case study is inspired by Phototourism [27] 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. 16 shows the task structure of this application. The cloud server is emulated by a Sun Ultra 20 workstation.

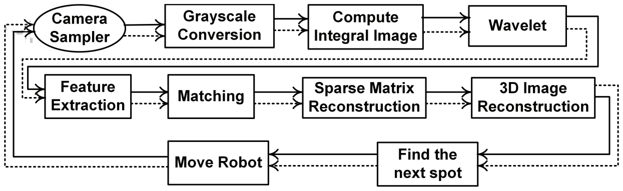
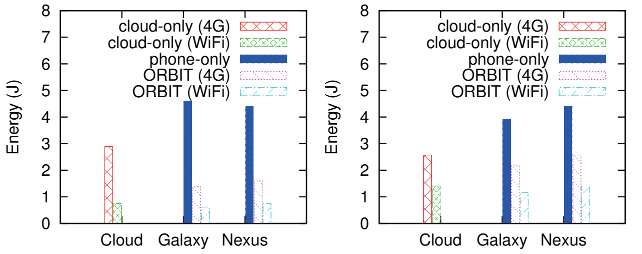
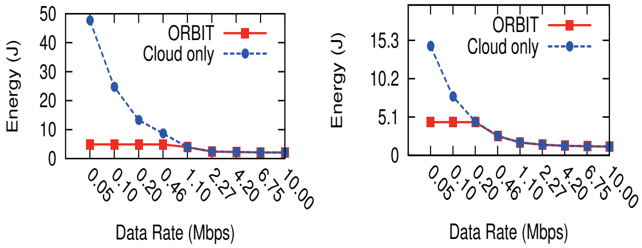


Figure 16: The block diagram of the Multi-camera 3D reconstruction application.



(a) House image (1280x722px) (b) kermit image (640x480px)

Figure 17: The results of various partition schemes.



(a) House image (1280x722px) (b) kermit image (640x480px)

Figure 18: Impact of data rate on the task assignment and total energy consumption.

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 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 [23, 5] often do not address the two affecting factors.

Impact of network throughput: We then evaluate the impact of the communication data rate, R, on the task partitioning. Fig. 18 shows the smartphone energy consumption for Nexus S. We can see that for both images the energy consumption decreases when R increases. The reason is that, when ORBIT is connected to the cloud via a faster link the amount of time the smartphone needs to stay active is shorter. This figure also shows that the energy consumption of ORBIT converges to that of the cloud-only. This is due to the fact that as the data rates increases it becomes more energy-efficient to offload tasks to the cloud. Fig. 18(a) and Fig. 18(b) show that the convergence to the cloud-only scheme occurs at different rates due to the difference in image size results in different upload delays. Overall, these results show that ORBIT can exploit the different characteristics of network latency and task workload to minimize the system energy consumption.

7.3 Discussion

These case studies demonstrate the generality of ORBIT's design. In particular, the two 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 both case studies is the system power consumption is highly probabilistic. 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. A possible improvement would be to provide ORBIT runtime feedback regarding system performance. This would allow ORBIT to optimize the wiring of tasks and priorities of tasks to reduce power consumption. Such runtime adaptation is readily supported by ORBIT due to its flexible task partition and dispatch framework.

Case study 2 suggests the communication delay between tiers and the input data play important roles in the partitioning result. For example, when the 3G or Wi-Fi network condition is of poor quality the tasks may not be offloaded to the cloud if the application imposes a tight delay bound. Moreover, advanced sensing tasks like image processing often have highly variable execution time which validates the online task partition framework adopted by ORBIT.

8. CONCLUSION AND FUTURE WORK

This paper presents 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. The results of the first two case studies are presented in the paper and the result of the last one can be found in a technical report [21]. 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. Future plans include evaluating ORBIT with additional embedded sensing applications, extending its data processing library and using it in large-scale deployments.

Acknowledgments

The authors thank the shepherd Dr. Jack Stankovic and anonymous reviewers for providing valuable feedbacks to this paper. This work was supported in part by U.S National Science Foundation under grants CNS-1218475, OIA-1125163, and CNS-0954039 (CAREER).

9. REFERENCES

[1] Arduino board. http://www.arduino.cc.

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

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

[4] Cloud robotics. http://goldberg.berkeley.edu/cloud-robotics/.

[5] 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 *MobiSys*, 2010.

[6] 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 *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[7] 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 *IPSN*, 2011.

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

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

[10] 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.

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

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

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

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

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

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

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

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

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

[20] Marvell sheevaplug. www.plugcomputer.org.

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

[22] Nasa phonesat project. http://open.nasa.gov/plan/phonesat/.

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

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

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

[26] Raspberry pi. http://www.raspberrypi.org.

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

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

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

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

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