DroidSense: A Toolkit for Smartphone-Based Data-Intensive Embedded Sensing Systems

Abstract

Owing to the rich processing, multi-modal sensing, and versatile networking capabilities, smartphones are increasingly used in an emerging class of 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 functionalities and embedded programming support. This paper presents DroidSense, a toolkit for smartphone-based platform to build data-intensive embedded sensing systems. DroidSense features a tiered architecture in which smartphone is interfaced extBoard, an energy-efficient peripheral board. The extBoard accomplishes time-critical tasks, such as high-rate sensor sampling and lightweight signal processing, and activates the smartphone to execute computation-intensive tasks in an on-demand manner. In particular, by intelligently dispatching the computation tasks to either the extBoard or to the smartphone, DroidSense minimizes the system energy consumption, subject to upper-bounded processing delays. DroidSense also includes a signal processing library that supports such task partition through new embedded programming primitives. Extensive case study evaluation and a pilot one-week field deployment validate the design of DroidSense.

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 people-centric in that the smartphone utilizes on-board sensors to sense people and characteristics of their contexts. Moreover, several applications leverage the power of crowds to collect data from [22] or coordinate sensing/computation among [29] a large number of participating phone users. The major challenges posed by these applications include inference of user location/context, incentive, privacy and security.

In this paper, we consider a new emerging class of smartphone-based embedded sensing systems. For example, in the Floating Sensor Network project [6], smartphone-equipped drifters can be deployed rapidly to collect real-time data about the flow of water through a river. The GPS-enabled smartphone allows the drifter to measure volume and direction of water based on real-time location and transmit the data back to the server through cellular networks. Smartphones have also been employed for monitoring the earthquakes [22] and operating miniature satellites [13]. Recently, there is increasing interests of integrating smartphones with robots. Such “cloud robots” leverage a plethora of sensors available on smartphone 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, smartphone has several salient advantages that make it a promising platform for aforementioned embedded applications, which include multiple short/long-range network interfaces (Bluetooth, WiFi, 3G/4G), various integrated sensors (GPS, accelerometer, gyro, compass, light, and etc.), abundant computation and storage resources, and user-friendly development environment. Moreover, the price of smartphones has been dropping drastically in the last decade. Many low-end Android phones (e.g., LG Optimus Net with 800MHz CPU and 2GB memory) cost only about US$100 [10]. These features make it possible to build a large-scale embedded sensing systems based on inexpensive off-the-shelf smartphones.

However, several system challenges must be addressed before smartphone can be used as a generic platform for embedded sensing systems. First, the smartphone power management schemes, designed to make phone last a few days, are ill-suited for many embedded sensing applications that must sustain a long period 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). The continuous sensor sampling can prevent the smartphone from entering sleep state, leading to depletion of the battery in a few hours. Moreover, the current major smartphone operating systems do not provide real-time functionalities, such as bounded interrupt handling delay, constant sampling rate, and precise timestamping, which are crucial to many embedded sensing applications. For instance, our measurements show that detecting a pulse signal through USB interface of Android phones suffers an unpredictable delay up to 5ms, which makes it impossible to achieve high constant sampling rate. Lastly, the programming environment of smartphone, although simplifies 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 native real-time signal processing libraries and communication/control primitives for peripheral sensors.

In this work, we take the first step toward addressing these challenges. We present DroidSense, a toolkit for smartphone-based platform to build data-intensive embedded sensing systems. DroidSense is designed for a tiered hardware architecture in which smartphone is interfaced with an energy-efficient peripheral board (referred to as extBoard). A number of such hardware boards are currently available, such as Arduino [3] and IOIO [8], which can readily integrate various accessories such as external sensors to Android phone via micro-USB interface. We propose a novel partition-based power management strategy — the time-critical tasks like high-rate sensor sampling are ex-
executed on the extBoard while computation-intensive tasks are offloaded to the phone. As a result, the phone can sleep for most of the time and only wake up occasionally to process the collected data. DroidSense integrates a novel task partitioning framework that can intelligently split the tasks between extBoard and phone to minimize the total energy consumption while meeting the specified time deadlines. DroidSense also integrates an extensible and composable library of algorithms that are optimized for real-time signal processing. This library also provides a new programming primitive called connection that enables task partitioning and allows programmers to easily hook different signal processing functions together to build sophisticated sensing applications. Owing to these features, DroidSense is a powerful toolkit to build a wide spectrum of energy-efficient data-intensive embedded sensing applications. Extensive case study evaluation validates the design of DroidSense. Moreover, a pilot deployment of six DroidSense nodes in an active volcano monitoring project demonstrates the feasibility and advantages of smartphone-based embedded sensing platforms.

2 Related Work

Mobile sensing based on smartphones has recently received extensive research. Most studies focus on identifying the human-centric contexts. Early studies such as SeeMon [25] and Orchestrator [26] deal with the challenges of limited computation and storage resources on the hand-held devices to support multiple concurrent sensing applications. SymPhoney [18] coordinates the resource use of concurrent context sensing applications on a smartphone. Kobe [20] is a tool for developing context classifiers based on sensor readings. Different from all above studies that focus on human-centric context sensing, this work targets data-intensive embedded sensing applications. Extensive case study evaluation validates the design of DroidSense. Moreover, a pilot deployment of six DroidSense nodes in an active volcano monitoring project demonstrates the feasibility and advantages of smartphone-based embedded sensing platforms.

2 Related Work

Mobile sensing based on smartphones has recently received extensive research. Most studies focus on identifying the human-centric contexts. Early studies such as SeeMon [25] and Orchestrator [26] deal with the challenges of limited computation and storage resources on the hand-held devices to support multiple concurrent sensing applications. SymPhoney [18] coordinates the resource use of concurrent context sensing applications on a smartphone. Kobe [20] is a tool for developing context classifiers based on sensor readings. Different from all above studies that focus on human-centric context sensing, this work targets data-intensive embedded sensing systems that capture and process high-rate, continuous physical signals, and must operate for extended time periods based on limited power supplies.

Recently, smartphones have been used in a number of embedded sensing applications. In [22], smartphones are used to build an earthquake early warning system. The geographically distributed smartphones detect rare earthquake events via on-board accelerometers and send to a server for centralized processing. In the Floating Sensor Network project [6], smartphone-equipped drifters are deployed to monitor waterways and collect real-time volume and direction of flow of water based on phone GPS. The NASA PhoneSat project [13] aims to build low-cost satellites based on Android smartphones. Controlled by smartphone, such small satellites could take various tasks such as earth observation and space debris tracking. PhoneSat supplements the smartphone with an Arduino board that reboots the smartphone if it crashed due to strong cosmic rays. Several recent research efforts focus on building “cloud robots” [5] that integrate smartphones with robots. The phone’s built-in sensors are used for sensing and navigation. The smartphone can also offload computation-intensive tasks like image and voice recognition to the cloud. Motivated by these pilot studies, DroidSense provides a generic toolkit for developing a wide spectrum energy-efficient, data-intensive embedded sensing applications.

Remote process execution was first studied as a computing paradigm in distribution systems. Spectra [23] allows programmers to specify program partitioning plans given application-specific qualities of service requirements. At run time, Spectra monitors the network connectivity and chooses the partitioning plan that maximizes a user-defined utility function. Similar remote execution concepts have also been adopted for mobile computing environments. Chroma [19] applies the principles of Spectra and aims to reduce the burden on manually defining the detailed partitioning plans. MAUI [21] enables fine-grained offload of the programs on a smartphone to the cloud to prolong the battery lifetime. Medusa [29] is a crowd-sensing programming framework that features a distributed runtime system to coordinate the execution of tasks between smartphones and cloud. Different from these remote execution schemes, DroidSense schedules the executions of signal processing algorithms on the smartphone and extBoard to maximize the battery lifetime subject to the application-specific real-time requirements.

3 Motivation and Design Objectives

This section first discusses the motivation of using smartphone as a platform for data-intensive embedded sensing. We then discuss the design objectives of DroidSense.

3.1 Motivation

The mote-class sensing platforms [11] have been widely adopted by wireless sensor network applications in the past decade. They are typically equipped with the TI MSP430 and Atmel ATmega processors [11] and 802.15.4 wireless transceivers [28]. These platforms are highly power-efficient and run TinyOS—a lightweight operating system (OS) extensively optimized for long-term, low-duty-cycle operations. However, due to the limited processing and storage capabilities (a few MHz MCU, a few KB RAM, and typically less than 1MB flash), they are ill-suited for high-sampling-rate sensing applications. Several high-end TinyOS-based mote platforms such as Imote2 [28] and SunSPOT [17] have been used in several data/computation-intensive applications such as structure health monitoring [24] and volcano monitoring [31]. However, they are often based on non-extensible designs. For instance, limited flash memory prohibits logging raw sensor data. Moreover, the limited production has led to relatively high unit costs (typically several hundred dollars without any on-board sensors), which impedes their adoption in large-scale sensing applications. Recently, several single-board computers such as Gumstix [7], SheevaPlug [12], and Raspberry Pi [15] have been used in embedded applications. They are equipped with rich processing and storage capabilities and can be packaged into a fist-size box or even a credit-card-size board [15]. However, their design is not particularly optimized for low-power embedded sensing. They run general-purpose Linux distributions that are not customized to reduce power consumption. Moreover, with no on-board sensors and wireless interfaces, they need to be equipped with various accessories for different sensing applications.

Table 1 compares the representative products of wireless mote, embedded computer, and smartphone platforms. The advantages of smartphone are summarized as follows:

<table>
<thead>
<tr>
<th>Product</th>
<th>Wireless Mote</th>
<th>Embedded Computer</th>
<th>Smartphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Efficiency</td>
<td>Low</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Processing Power</td>
<td>Low</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Storage</td>
<td>Limited (MB)</td>
<td>GBs</td>
<td>TBs</td>
</tr>
<tr>
<td>Development Cost</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Battery Lifetime</td>
<td>Low</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>Compute Intensity</td>
<td>Low</td>
<td>Moderate</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Representative Products

This table summarizes the relative advantages of different platforms in mobile sensing. Smartphones offer the highest power efficiency and compute intensity, making them ideal for data-intensive sensing applications.
### Table 1. Comparison of various sensing platforms.

<table>
<thead>
<tr>
<th>Type</th>
<th>Wireless mote</th>
<th>Embedded computer</th>
<th>Smartphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>TelosB</td>
<td>Imote2</td>
<td>Sheevaplug LG GT540</td>
</tr>
<tr>
<td>CPU freq.</td>
<td>8MHz</td>
<td>416MHz</td>
<td>1.2GHz 600MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>10KB</td>
<td>32MB</td>
<td>256MB 156MB</td>
</tr>
<tr>
<td>Flash</td>
<td>1MB</td>
<td>32MB</td>
<td>128MB 2MB</td>
</tr>
<tr>
<td>Sensor</td>
<td>light, humidity temperature</td>
<td>N/A</td>
<td>GPS, bat, cam</td>
</tr>
<tr>
<td>Networking</td>
<td>Zigbee</td>
<td>Ethernet, Wireless</td>
<td>3G, WiFi, Bluetooth</td>
</tr>
<tr>
<td>OS</td>
<td>TinyOS</td>
<td>Linux</td>
<td>Android</td>
</tr>
<tr>
<td>Programming</td>
<td>nesC</td>
<td>Any</td>
<td>Java, C/C++</td>
</tr>
<tr>
<td>Library/tool</td>
<td>built-in library and contrib</td>
<td>GNU</td>
<td>Android SDK</td>
</tr>
<tr>
<td>User interface</td>
<td>LED, user buttons</td>
<td>N/A</td>
<td>Touch screen</td>
</tr>
<tr>
<td>Power</td>
<td>≤70mW</td>
<td>1.8-300mW</td>
<td>2.3-7W 4.5mW</td>
</tr>
<tr>
<td>Price</td>
<td>$139</td>
<td>$299</td>
<td>$99  $115</td>
</tr>
</tbody>
</table>

The listed power consumption and price do not account for external sensors and wireless interfaces.

### Rich computation and storage resources

Compared with motes, smartphones have abundant computation and storage capabilities, which make them suitable for data-intensive sensing applications such as structural health monitoring [24] that process a large amount of data inside the network.

#### Multiple network interfaces

Most smartphones are shipped with multiple network interfaces including cellular (3G/4G), WiFi and Bluetooth. They can leverage existing WiFi/cellular infrastructures or use WiFi ad hoc, WiFi Direct, and Bluetooth to form a potentially multi-hop network.

#### Multiple sensing modalities

Most smartphones have onboard GPS, microphone, accelerometer, camera and ambient light sensor. Gyroscope, compass, and temperature sensor are increasingly available for mainstream smartphones. Moreover, external sensors can be integrated with smartphones using commercial-off-the-shelf (COTS) extension boards such as IOIO [8] and Arduino [3].

#### User-friendly programming environments

Modern smartphone OSes offer easy-to-use integrated development environments and extensive library support. In contrast, the application development for motes is based on the GNU toolchain that has a steep learning curve.

#### Interactive user interface

In contrast to the limited user interfaces of motes and embedded computers (e.g., LED indicators and buttons), the touch screen of smartphones can display the status and results of the sensing applications as well as receive user commands. This feature is particularly useful for program debugging during the development process and in situ control of the nodes.

#### Diversity and low cost

Due to the massive production, the price of smartphones has been dropping drastically (down to about $50 [4]). Moreover, the diverse configurations of different smartphone models offer the system designers significant flexibility to trade-off various factors such as hardware capability, power consumption, and cost.

### 3.2 Challenges and Design Objectives

Despite various advantages of smartphones, we face the following major challenges in building an embedded sensing platform based on COTS smartphones:

#### High power consumption

The OSes of smartphones are carefully optimized to extend battery time. For instance, the phone typically switches to deep sleep state in the absence of user activity to save energy. However, designed to adapt to user activities, such power management schemes are not suitable for untethered embedded sensing systems. For instance, if the smartphone samples sensors continually, its CPU cannot enter the deep sleep state to save energy.

#### Lack of real-time functionalities

Many sensing applications have stringent real-time requirements, such as constant sampling rate and precise timestamping. However, most modern smartphone OSes are not designed for meeting these real-time requirements. For instance, Android only allows programmers to choose one of four descriptive levels of sensor sampling rate, rather than the exact rate. Moreover, the sensor sampling can be delayed by the CPU tasks with higher priority like Android system services or UI drawing.

#### Lack of embedded programming support

The programming environment of smartphone is designed to facilitate the development of networked, user-centric mobile applications. It simplifies many programming tasks in the life cycle of embedded systems such as debugging, remote data logging, visualization, and software maintenance. However, it lacks important embedded programming support such as native real-time signal processing libraries and primitives for controlling and communicating with peripheral accessories like external sensors.

In this paper, we present DroidSense, a toolkit for Android smartphone-based embedded sensing platforms. DroidSense is designed to meet three objectives. First, DroidSense should only rely on the “out of the box” functionalities of COTS smartphones. For instance, DroidSense should be installed on the phone as an App and on the extBoard as an application package, without requiring low-level customization or rooting the device. This not only minimizes the burden on the application developers, but also ensures the compatibility with the diverse smartphone models. Although it is possible to apply patches to the Linux kernel of Android to gain certain real-time capabilities [16], such changes are only compatible with certain devices and likely make future system upgrades difficult. Second, DroidSense should provide necessary embedded programming support, such as a rich library of signal processing algorithms, which have well-defined interfaces and can be easily composed to build sophisticated sensing applications. More importantly, the library should be extensible such that the developers can easily add application-specific signal processing algorithms or port legacy libraries. Third, DroidSense should automatically optimize the power consumption of a sensing application while meeting the specified deadlines.

### 4 System Overview

#### 4.1 Hardware Composition

A DroidSense node is composed of an Android smartphone and an extBoard (e.g., IOIO [8] and Arduino [3]). The extBoard is equipped with a low-power MCU (e.g., ATMega2560 with 16 MHz frequency and 8 KB RAM), multiple analog-to-digital (A/D) and digital channels. Various analog sensors can be easily connected to the A/D pins of the extBoard. Therefore, the extBoard provides a hardware abstraction of accessories being integrated with the system.
and a unified way to access them. The extBoard, powered by external batteries, is connected to the smartphone using a USB cable for both communication and charging the on-phone battery. The extBoard has native development libraries (e.g., Arduino SDK [3]) that enable the smartphone to access the accessories through the USB cable. Fig. 1 shows a DroidSense node connected with a seismometer, which is used for real-time volcano monitoring (cf. Section 8).

4.2 System Architecture

DroidSense is designed as a multilayered system as illustrated in Fig. 2. The major components of DroidSense include: (1) an embedded program running on the extBoard, which samples the attached sensors and performs lightweight tasks, (2) Task Controller, an Android App running on the smartphone, executes the computation-intensive tasks and coordinates the operations of smartphone and extBoard, (3) a Task Partitioner that dynamically optimizes the dispatch of tasks to smartphone or extBoard to minimize energy consumption subject to processing delay, and (4) a library of signal processing algorithms, which can be easily connected to compose advanced sensing applications. These four components are integrated to enable the efficient collaboration between the smartphone and extBoard to accomplish real-time, energy-efficient sensor sampling and signal processing. The unique features of DroidSense include:

Easy embedded application development: As the signal processing algorithms in the DroidSense library implement unified interfaces, they can be easily composed into various advanced sensing applications. The library provides a programming primitive called connection that allows programmers to specify application composition in an XML file. In particular, each algorithm can be executed on both the smartphone and extBoard, which enables the task dispatch to achieve tight deadlines and reduce energy consumption.

Efficient task partitioning: In real-time data-intensive sensing applications, collected sensor data must be processed before tight deadlines. To meet these deadlines, the computation-intensive tasks are assigned to the smartphone that has abundant processing capabilities. However, due to the high power consumption of smartphone, it is desirable to reduce its use to extend system lifetime. We formally formulate a constrained optimization problem that aims to minimize the energy usage of the smartphone subject to a processing delay bound on time-critical tasks. DroidSense solves this problem and obtains the optimal task dispatch plan.

Online phone-extBoard coordination: At run time, the Task Controller instantiates the tasks and executes them by following the task dispatch plan. The extBoard runs low-level functions such as sensor sampling and timestamping as well as lightweight signal processing tasks. The smartphone sleeps for most of the time to save energy, and is only woken up by the extBoard to run assigned computation-intensive tasks. By the effective coordination between the smartphone and extBoard, DroidSense fully exploits the heterogeneity in their power/delay profiles to construct power-efficient data-intensive sensing applications.

5 Measurement-Based Power and Latency Profiling

To use smartphone as the base platform for embedded sensing, it is important to understand the characteristics of its power consumption and latency. This section presents a measurement-based profiling of the power consumption and latency of smartphone and extBoard. The empirical models derived in this section will be used by the task partitioning framework to optimize system energy consumption.

5.1 Power Profiling

Our measurement study focuses on profiling CPU of smartphone because computation is typically the dominant source of power consumption for data-intensive sensing tasks. However, the power consumption models of other components (e.g., radio) can be easily integrated into DroidSense. We measure the current draw of several Android smartphones using an Agilent 34411A Multimeter [1]. We observed similar CPU state transitions and power consumption characteristics. Fig. 3(a) shows the current draw in different states of a Samsung Nexus S. Initially, the smartphone is in sleep state, and hence draws little current (less than 5 mA). At the 5th second, the extBoard requests the smartphone to execute the FFT algorithm. Upon receiving the request, the phone first acquires the wake lock [2], which is a mechanism of Android to prevent the phone from going...
to sleep regardless of the states of the phone peripheries (e.g., screen). At the 25th second, the FFT algorithm 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. 3(b) shows the zoomed-in view of two transitional phases. We refer to them as wake-up and tail phases, which last about 200 ms and 755 ms, respectively. There are also two spikes in Fig. 3(a), which are caused by the communication between the phone and extBoard. Since these spikes are very short with limited current draw, the energy consumption of them is negligible. Based on these results, we classify the CPU state as sleep, wake-up, active and tail. Fig. 4 shows the state machine that describes the state transition model of smartphones.

We now describe an empirical energy consumption model based on our measurement results. Let $P_s$ and $P_a$ denote power consumption in the sleep and active states, respectively, which are constants. Let $t_s$ denote the time that the phone stays in the sleep state, which is a variable. Therefore, the energy consumption during the sleep state is $E_s = P_s t_s$. Suppose the extBoard requests the smartphone to execute a series of algorithms, which is denoted by set $A$ and the execution time of algorithm $i$ is $t_i$ where $i \in A$. The processing energy consumption is $E_w = P_a \sum_{i \in A} t_i$. We assume that the wake-up and tail states consume a total energy of $E_w$ and $E_t$, and last for $T_w$ and $T_t$ seconds, respectively. We measure the constants $P_s$, $P_a$, $E_w$, $E_t$, $T_w$, and $T_t$ using three smartphones of different models and the results are summarized in Table 2. From the table, the maximum variation is 5%. Therefore, it is safe to assume that they are constants.

Fig. 5 shows the current draw of an Arduino board in different CPU states. Its average current draw in active and idle states are 90 mA and 66 mA. As opposed to the smartphones, the transitional states for Arduino are very short (in the order of $\mu$s) and hence their energy consumption is negligible.

### 5.2 Latency Profiling

The execution times of the tasks determine their energy consumption and highly affect the real-time performance of the application. Fig. 6 plots the execution times of various signal processing algorithms on an Arduino and a Nexus S smartphone, versus the length of input signal. We have two important observations. First, the latencies on the extBoard and smartphone are in the orders of seconds and milliseconds, respectively. As they have comparable active powers as shown in Section 5.1, the smartphone can process the signals with much less energy and latency. Second, the execution time increases with the length of input signal. However, it is difficult to find a consistent model to characterize the latencies of algorithms on the extBoard and smartphone. For instance, the slowest algorithms on them are different in spite of that the four tested algorithms have the same C++ implementations. Therefore, the latencies of algorithms must be measured to ensure the real-time performance of the application as well as reduce energy consumption.

### Table 2. Energy model parameters.

<table>
<thead>
<tr>
<th>Unit</th>
<th>LG G1340</th>
<th>Galaxy Nexus</th>
<th>Nexus S</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_s$ mW</td>
<td>4.07 ± 0.74</td>
<td>52.17 ± 2.8</td>
<td>6.66 ± 3.9</td>
</tr>
<tr>
<td>$P_a$ mW</td>
<td>568.7 ± 6.06</td>
<td>779.59 ± 13.1</td>
<td>1079.29 ± 24.05</td>
</tr>
<tr>
<td>$E_w$ mJ</td>
<td>75.7 ± 0.7</td>
<td>18.75 ± 0.93</td>
<td>271.3 ± 2.1</td>
</tr>
<tr>
<td>$E_t$ mJ</td>
<td>59.9 ± 0.1</td>
<td>6.47 ± 0.46</td>
<td>67.2 ± 1.1</td>
</tr>
<tr>
<td>$T_w$ ms</td>
<td>1100 ± 35</td>
<td>1200 ± 52</td>
<td>752 ± 22</td>
</tr>
<tr>
<td>$T_t$ ms</td>
<td>50 ± 2</td>
<td>10 ± 0.5</td>
<td>199 ± 7</td>
</tr>
</tbody>
</table>

* The errors represent standard deviations

Figure 5. The current draw profile of an Arduino board.

Figure 6. Execution time of signal processing algorithms.
5.3 Timing Accuracy Profiling

Timing accuracy is critical for many sensing applications. For instance, acoustic or seismic source localization in earthquake and volcano monitoring [27] typically requires time stamps of sensor samples to have sub-millisecond-level precision. In this section, we measure the accuracy of \textit{timer} and \textit{event timestamping} of Android smartphones and discuss the impact on the design of DroidSense. Timer is commonly used to implement constant-rate sensor sampling and its accuracy determines the maximum sampling rate that can be supported. We are also interested in the delay of timestamping an external event, which may be triggered by GPS receiver or sensor connected to the phone through USB.

Our measurements are conducted using LG GT540, Galaxy Nexus, and Nexus S Android smartphones. All the phones exhibited similar level of timing variability in our experiments and we only discuss the results of LG GT540 here. We tried the \texttt{sleep} functions of C (via NDK) and Java, as well as \texttt{java.util.Timer}. All of them have poor timing performances. Fig. 7 plots the distribution of the interval between two software interrupts generated by C’s \texttt{sleep} with an aimed sampling rate of 100Hz. We can see that the distribution has a long tail. Although most intervals are close to 10ms, the maximum one is above 100ms. We then measure the delay between the time instant when a pulse signal is received by a digital pin of Arduino (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 5ms. These results suggest that the Android system has poor timing accuracy, which makes it impossible to implement high-constant-rate sensor sampling or precise event timestamping. In contrast, our measurements show that the timing error of Arduino is no greater than 12µs, due to the availability of hardware timers and efficient interrupt handling. Therefore, DroidSense partitions the tasks in a sensing application based on their time-criticality and energy consumption, and only executes real-time tasks on extBoard.

6 System Design

This section presents the design of DroidSense to meet the objectives discussed in Section 3.2.

6.1 Signal Processing Library

DroidSense provides a library of signal processing algorithms, which includes the commonly used signal processing algorithms ranging from basic statistical operations like mean removal to advanced signal transforms. With these well tested functions, the developers can quickly construct their sensing applications by simply connecting different building blocks via a configuration file. This library has two main design objectives. First, it is hierarchical and extensible so that developers can easily add more algorithm implementations or port legacy signal processing libraries. More importantly, it is designed to be compatible with both the smartphone and extBoard, and enable automatic tasks execution manipulation (cf. Section 6.3) and optimization (cf. Section 6.3) on the DroidSense platform.

6.1.1 Hardware Compatibility

As shown in Section 5.2, smartphone and extBoard have substantially different power consumption and latency profiles. DroidSense exploits this heterogeneity by partitioning sensing tasks and scheduling their executions on different platforms. To enable automatic execution manipulation and optimization, DroidSense requires the library to be executable on both smartphone and extBoard. An alternative approach would be developing two separate versions of the library for smartphone and extBoard. However, this likely incurs significant burden on application developers when extending the library with new algorithms.

We choose C++ as the programming language to implement all algorithms in the library. This allows many legacy C/C++ implementations of commonly used algorithms (e.g., FFT) to be easily ported to DroidSense. On the extBoard, the signal processing functions used by an application are compiled and linked into an executable that is then loaded into the program flash memory. As most low-end extBoards such as Arduino are not equipped with floating-point unit, we replace the operations with intensive floating-point arithmetic by fixed-point versions. For the smartphone, the C++ implementations of the algorithms are compiled into dynamically loadable libraries (i.e., *.so files) and wrapped by Java Native Interface (JNI) [9]. The wrapping allows the Task Controller (cf. Section 6.4) to load the corresponding *.so files and execute the algorithms. In Section 7.2, we will evaluate the overhead incurred by the wrapping.

6.1.2 Hierarchy and Extensibility

By leveraging the object-oriented programming paradigm of C++, the algorithms can be organized into a hierarchical library as shown in Fig. 2. An algorithm is encapsulated into a class, which implements unified input/output (I/O) interfaces. An instantiated class is referred to as a task. Any class in the library is derived from the abstract class

<table>
<thead>
<tr>
<th>Table 3. Members of the \texttt{ds_base_task} class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>assignment</td>
</tr>
<tr>
<td>process</td>
</tr>
<tr>
<td>meta_record</td>
</tr>
<tr>
<td>frameable</td>
</tr>
</tbody>
</table>

A task class overrides this function with the implementation of the signal processing algorithm (e.g., FFT). The I/O of a task is composed of multiple signal segments. A signal segment in the I/O is also referred to as a pin. The I/O is defined by the class \texttt{ds\_io}, which contains an array (specifically, std::vector<int>) and a list of signal lengths. The
signal segments are concatenated in the array. Therefore, they can be accessed according to the list of signal lengths. In DroidSense, the sensor samples in a signal segment are stored using 32-bit signed integers (i.e., int). Therefore, we consistently use int as the input/output data type. By this design, all tasks have uniform interfaces and two pins can be easily connected by the Task Controller without type casting. The output of an algorithm can be real numbers, e.g., the cross-correlation of two signal segments. In such a case, the results are stored in fixed-point numbers. The argument ds_frame_arg in the above virtual function is used for data framing and will be explained in Section 6.3.4. Each task class is associated with a list of meta records, which are the execution times on various smartphones and extBoards. They will be used by the Task Partitioner (cf. Section 6.3) to optimize the energy consumption of the sensing application.

The current prototype of DroidSense library includes 15 commonly used algorithms that have been fully tested on Arduino [3] and a number of different smartphones. The application developer can also extend the library with new signal processing algorithms by deriving from the abstract class ds_base_task. A general guideline is that a single task accomplishes a non-divisible signal processing function. This facilitates the fine-grained energy optimization of the application because the DroidSense Task Partitioner aggressively selects the execution venue of each task to minimize the system energy consumption. The application developer also needs to measure the execution times of the added tasks on the smartphone and extBoard. This can be easily done by using the timing tool provided by DroidSense.

6.2 Building Applications from DroidSense Library

As the tasks in DroidSense library have uniform I/O interfaces, they can be connected to build applications. In this section, we describe the concept of task connection and the XML schema defined by DroidSense to specify how the tasks are connected to build applications.

6.2.1 Connections

Two tasks can be connected by a directed link, from source task to destination task. There are two connection primitives: data connection and execution connection.

Data connection: The destination task of a data connection takes an output signal segment of the source task as an input signal segment. Therefore, the connected pins must have the same signal length. The tasks and the associated data connections can be described by a directed acyclic graph, where tasks are nodes and data connections as edges. This tree is referred to as execution tree, and denoted by T. The root of T is the entry task of the application.

To simplify the design of Task Partitioner and Task Controller, DroidSense poses the following requirements. First, for a branching execution connection, the output pin of the source task must be a scalar, and the condition is the union of the ranges of the scalar. Second, for multiple branching execution connections that have the same source task, their conditions are exclusive. Therefore, only one destination task will be executed. Third, for a data connection, its source task must be an ancestor of its destination task in the execution tree. This ensures that the input to the destination task of the data connection must be available before its execution.

The connection primitives defined by DroidSense allow developers to build complicated sensing applications. We now discuss how to use the connections to build a broad class of event detection applications (e.g., [22, 24, 27, 32]). Initially, the raw signal goes through a series of preprocessing algorithms (e.g., calibration, mean removal, and low-pass filtering) that are connected using sequential execution connections. Various feature extraction algorithms (e.g., signal energy, FFT, zero-crossing count, and wavelet) are then applied to the pre-processed signal. Note that these algorithms may take the same pre-processed signal as input, and hence they are logically parallel. That is, the final sensing result is independent of the order of their executions. Therefore, the developer can connect them in any order using sequential execution connections, and add a data connection from the last preprocessing task to each of them. The task that performs the actual event detection takes the extracted features as inputs. Therefore, data connections can be added between each feature extraction task and the detection task.

6.2.2 Application Specification

An application is defined by an XML file (referred to as application specification) that specifies which tasks are used and how they are connected. We have defined an XML schema and developed an interpreter to read and validate the application specification against the schema. The interpretation result will be used by both the Task Partitioner and Task Controller. From now on, we will use a running example to illustrate how tasks are connected to build an application, as well as the automatic execution optimization and manipulation in later sections. Fig. 8 shows the dataflow graph and

![Figure 8. An example DroidSense application.](image-url)
execution tree of the example application. This application is composed of 12 tasks (i.e., $T_1$ to $T_{12}$). Each task is described as an XML element in the application specification. For instance, $T_2$ is described as:

\[
\text{<task ID="2" name="T2" insize="100" outsize="100,1" />}
\]

where the insize and outsize specify the lengths of the I/O signal segments. $T_2$ has an input pin and two output pins. The second output pin is a scalar, which will be used for a branching execution connection. The dashed arrows represent the data connections. For instance, $T_2$ takes the first output of $T_1$ as the input, which is described as:

\[
\text{<connect from="1" to="2" fromOut="0" toIn="0" />}
\]

where the from and to specify the IDs of connected tasks, fromOut and toIn specify the index of the I/O pins. The solid arrows represent the execution connections and the circles represent the branches. For instance, the execution connection between $T_2$ and $T_1$ is sequential and described as:

\[
\text{<sequence from="7" to="11" />}
\]

The execution connections between $T_1$, $T_2$, and $T_3$ are branching. The second output pin of $T_1$, a scalar, is used to compare against zero. If it is non-negative, $T_2$ will be executed; otherwise, $T_3$ will be executed. Therefore, these two connections are described as:

\[
\text{<branch from="1" to="2" fromOut="1" condition="[0,+\infty)" />}
\]
\[
\text{<branch from="1" to="3" fromOut="1" condition="[-\infty,0)" />}
\]

In the rest of this paper, we will follow the representations of connections in Fig. 8, i.e., a solid arrow is an execution connection, a dashed arrow is a data connection, and a circle is a branch. Fig. 13 in Section 7.2 shows the detailed XML specification of a case study used in our evaluation.

### 6.3 Task Partitioning

A key design objective of DroidSense is to minimize the system energy consumption while meeting the deadlines of sensing applications. DroidSense adopts a novel task partitioning framework that exploits the heterogeneity in power consumption/latency profiles of smartphone and extBoard. In particular, DroidSense schedules the execution of a task on smartphone or extBoard based on its time-criticality and energy consumption. This section first discusses the power management strategies of DroidSense, and then describes the partitioning algorithms for both sequential and branching tasks. Finally, we discuss an energy consumption optimization heuristic based on fine-grained data framing.

#### 6.3.1 Smartphone/extBoard Power Management

Our measurement-based study in Section 5 shows interesting heterogeneity in the profiles of power consumption/latency of smartphone and extBoard. Although the smartphone is more energy-efficient in executing compute-intensive tasks, it is ill-suited for time-critical tasks like high-rate sampling because of its poor timing accuracy. Motivated by this observation, DroidSense employs different power management strategies for smartphone and extBoard. 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. Note that the signal segment received in the current sampling duration will be processed by the extBoard and smartphone in the next sampling duration. 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 extBoard are often subject to stringent delay bounds. Different from extBoard, the smartphone adopts an on-demand sleep strategy in which it remains asleep unless activated by extBoard. Fig. 9 illustrates the duty cycle of extBoard and on-demand sleep schedule of smartphone.

#### 6.3.2 Partitioning Sequential Execution

As discussed in Section 6.3.1, the extBoard has a fixed duty cycle and hence consumes relatively constant energy. Therefore, DroidSense aims to minimize the total energy consumption of smartphone, subject to upper-bounded extBoard processing delay. To this end, DroidSense employs a novel task partitioning framework that assigns the execution of each task to either smartphone or extBoard. In this section, we consider a sensing application consisting of $n$ tasks (denoted by $T_1, \ldots, T_n$) that are sequentially connected. Specifically, the execution tree is simply a list $T = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$. Formally, the task partitioning problem can be formulated as follows:

**Task Partitioning Problem.** For the sequential execution $T = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, the Task Partitioner finds a 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 that the extBoard processing delay in a sampling duration (denoted by $\tau$) is upper-bounded by $D$, where $I_1 = 0$ and $I_i = 1$ represent that $T_i$ is executed on the phone and extBoard, respectively.

The processing delay upper bound $D$ is typically set according to timeliness requirements of application, e.g., the constant rate of real-time sensor sampling. As the sensor sampling and time-stamping introduces little overhead (cf. Section 6.4.2), it is safe to set $D$ to slightly smaller than the sampling duration. We now derive $E$ and $\tau$.

The execution times of $T_i$ on the smartphone and extBoard are denoted by $t_{ia}$ and $t_{ib}$, respectively. Let $P$ denote the power consumption, where the superscripts ‘p’ and ‘b’ represent smartphone and extBoard, and the subscripts ‘a’ and ‘s’ represent active power and sleep power. Let $t_e$ denote the latency of downloading/uploading a data unit from/to the phone to/from the extBoard, $I_i$ the number of input pins of $T_i$, and $s_{i,j}$ the signal length of the $j$th input pin of $T_i$. We define a function $s(i,j)$ based on the dataflow graph $G$, which returns the ID of the source task that is connected with the $T_i$’s $j$th input pin by a data connection.

We now analyze the smartphone energy consumption and...
the extBoard processing delay for $T_i$, which are denoted by $e_i$ and $\tau_i$. The analysis accounts for not only the processing overhead (energy consumption and delay), but also the overhead of smartphone’s state transitions and copying the input data between extBoard and smartphone. Let $e_i'$ denote the smartphone energy consumed by task execution and state transition, and $e_i''$ denote the smartphone energy consumed by copying the input data of $T_i$ between extBoard and smartphone. Therefore, $e_i = e_i' + e_i''$. Let $\tau_i'$ and $\tau_i''$ denote the extBoard processing delays caused by task execution and copying the input data. Therefore, $\tau_i = \tau_i' + \tau_i''$. Note that as $\mathcal{T}$ is executed cyclically, for ease of presentation, we add a virtual task $T_0$ in the following analysis, where $I_0$ always equals $I_n$. We first derive $e_i'$ and $\tau_i'$ as follows:

**Case 1** ($I_{i-1} = 1$ and $I_i = 1$): As the phone sleeps when the extBoard executes $T_{i-1}$ and $T_i$, $e_i' = P_{i-1}^p t_i^b$ and $\tau_i' = t_i^b$.

**Case 2** ($I_{i-1} = 0$ and $I_i = 0$): As the phone is active for executing $T_{i-1}$ and $T_i$, $e_i' = P_i^p t_i^b$ and $\tau_i' = 0$.

**Case 3** ($I_{i-1} = 1$ and $I_i = 0$): The energy for executing $T_i$ on the smartphone is $P_i^p t_i^b$. Moreover, the smartphone needs to be waked up from the previous sleep state, causing the wake-up energy $E_w$. Therefore, $e_i' = P_i^p t_i^b + E_w$ and $\tau_i' = 0$.

**Case 4** ($I_{i-1} = 0$ and $I_i = 1$): Similar to Case 3, $e_i' = P_i^p t_i^b + E_i$ and $\tau_i' = t_i^b$. Note that as the smartphone switches to the sleep state, $e_i'$ includes the tail energy $E_t$.

Denote $E' = \sum_{i=1}^n e_i'$ and $\tau' = \sum_{i=1}^n \tau_i'$. It is easy to verify that

$$E' = \sum_{i=1}^n P_i^p t_i^b + \sum_{i=1}^n |I_{i-1} - I_i| \cdot E_w + \frac{E_T}{2}, \quad \tau' = \sum_{i=1}^n t_i^b.$$  

We now analyze $e_i''$ and $\tau_i''$. If the tasks $T_i$ and $T_{s(i,j)}$ are executed at different venues, the input data to the $j$th input pin of $T_i$ need to be copied between the smartphone and extBoard,\(^1\) causing smartphone energy consumption of $P_i^p t_i^c l_{i,j}$ and extBoard processing delay of $t_c l_{i,j}$. Therefore,

$$e_i'' = \sum_{j=1}^{J_i} |I_{s(i,j)} - I_i| \cdot P_i^p t_c l_{i,j}, \quad \tau_i'' = \sum_{j=1}^{J_i} |I_{s(i,j)} - I_i| \cdot t_c l_{i,j}.$$  

Denote $E'' = \sum_{i=1}^n e_i''$ and $\tau'' = \sum_{i=1}^n \tau_i''$.

As a result, the total smartphone energy consumption and the total extBoard processing delay for processing the sensor data collected in a sampling duration are

$$E = E' + E'', \quad \tau = \tau' + \tau''.$$  

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 duration if $D$ is close to

---

\(^1\)Performance optimizations can avoid some data copying. For instance, in Fig. 8, suppose $T_1$, $T_2$, and $T_4$ are sequentially connected, $I_1 = 1$, $I_2 = I_4 = 0$. As the data from $T_1$ have been copied to the phone for $T_2$, they can be reused for $T_4$. However, these performance optimizations can greatly complicate the design of Task Controller and make the analysis intractable.

---

### Algorithm 1 Partitioning Branching Execution

1: sort all leaf nodes in $\mathcal{T}$ according to priority number ascendingly
2: for $x$ in the sorted leaf nodes do
3: $X = \{x\}, p = x\.parent$
4: repeat
5: if $p$ has not been assigned then
6: $X = X \cup p, p = p\.parent$
7: else
8: $I = (p \equiv null ? null : p\.assignment)$
9: break
10: end if
11: until $p = null \lor$ the parent of root is null
12: run partitioning algorithm with $X$ and the preceding assignment $I$
13: update the assignment of each node in $X$
14: end for

---

the sampling duration. Therefore, the sleep energy consumption of the smartphone during the gap is negligible. Based on the above delay and energy models, the task partitioning problem is a constrained non-linear optimization problem. The nonlinearity comes from the formula of $E$ and $\tau$. Since the Task Partitioner runs offline, a brute-force search can be used to solve the problem. As the number of tasks in a sensing application is typically small, our measurements in Section 7.1 show that the brute-force search introduces little overhead even if the Task Partitioner is executed periodically by the smartphone (c.f., Section 6.3.3). For instance, its execution time is less than 10 ms when there is up to 20 tasks.

6.3.3 Partitioning Branching Execution

In this section, we discuss how to partition the application that contains branches. We continue to use the running example in Fig. 8 to illustrate our approach. Different from sequential tasks, a key challenge here is that the task partitioning solution that is optimal for all branches may not exist. We now use a part of Fig. 8 as an example, which only includes $T_1$, $T_2$, and $T_3$. 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 venues (smartphone and extBoard) in them. DroidSense adopts a priority-based approach to resolve the potential conflicts.

In the execution tree of Fig. 8, 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 6.3.2 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., $T_1 \rightarrow T_2 \rightarrow T_3$), 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 only run the task partitioning algorithm for the residual path (i.e., $T_5 \rightarrow T_9$) with the assignment of $T_2$ in $S_1$. We apply this procedure to all other paths. Algorithm 1 shows
the pseudo code of the task partitioning algorithm for the applications with branches. At run time, the Task Controller executes the tasks according to the assignment set. For instance, in Fig. 8, if it decides to execute $T_3$ according to the result of $T_2$, it will dispatch $T_2$ according to $S_2$.

### 6.3.4 Data Framing

This section discusses an improvement to reduce the smartphone energy consumption, when the application contains computation-intensive tasks. Data framing approach splits a single task into multiple sub-tasks where each sub-task processes a portion of the original input signal. The resultant sub-tasks are fed to the Task Partitioner, together with other tasks. The benefit of data framing is two-fold. First, many signal processing algorithms have super-linear time complexity with respect to the length of signal. As a result, the sum of time costs of sub-tasks may be smaller than that of the original task. Second, it may improve task partitioning efficiency. A computation-intensive task is likely assigned to the smartphone due to its long processing delay. As sub-tasks have shorter execution time, some of them can be assigned to the extBoard to fully utilize its CPU time during the active period of a duty cycle, reducing the smartphone energy consumption. In summary, data framing leads to finer-grained task partitioning, and may achieve a better trade-off between system energy consumption and latency.

However, extra care has to be taken in data framing to ensure the correctness of signal processing. We now use the matrix-vector multiplication algorithm as an example to illustrate this issue. Many signal processing algorithms (e.g., various transforms and compressive sampling) are actually matrix-vector multiplications. The input is $x \in \mathbb{Z}^l \times 1$ and output is $y = Ax$, where $A \in \mathbb{Z}^{m \times l}$. Suppose $x$ is evenly split into $K$ sub-s signals, i.e., $x = [x_1; x_2; \ldots; x_K]$, where $x_k \in \mathbb{Z}^{l/K \times 1}$. The matrix $A$ is split into sub matrices accordingly, i.e., $A = [A_1, A_2, \ldots, A_K]$, where $A_k \in \mathbb{Z}^{m \times l/K}$. The $k$th sub-task computes $y_k = A_k x_k$, and the final result is $y = \sum_{k=1}^{K} y_k$. Although the $k$th sub-task also performs matrix-vector multiplication, it needs the global information $K$ and $k$ to select the right sub signal and sub matrix.

Motivated by this example, we define a data framing structure shown in Fig. 10. A task is frameable if it can be decomposed into multiple sub-tasks and the sum of sub-task output is identical to the original output. We have carefully analyzed the signal processing algorithms implemented in DroidSense library and extended with a data framing interface. At run time, if a task is decomposed, the Task Controller will instantiate multiple sub-tasks from the same class. However, the sub-task needs to know whether it is running in the data framing mode. This is achieved by passing a `ds_frame_arg` to the third argument of the sub-task’s `process` function, which includes whether the task is executed in data framing mode, as well as $k$ and $K$.

### 6.4 Task Controllers

This section presents the design of the Task Controllers at the smartphone and extBoard, respectively. The two Task Controllers collaborate to execute the sensing tasks according to the assignment computed by the Task Partitioner.

#### 6.4.1 Task Controller at Smartphone

Task controller at the smartphone is designed as an Android background service, which manipulates the execution of the tasks and communicates with the extBoard. When the application is launched, the Task Controller acquires the wake lock, creates the instances of the tasks in $T$, and allocates the buffers for all inputs and outputs. Two pins that are connected in the dataflow graph $\mathbb{G}$ share a memory area. After this initialization phase, the Task Controller sends a start message to the extBoard, releases the wake lock, and switches to sleep state. The extBoard can activate the smartphone by reinitializing the USB communication with the smartphone. The activated Task Controller then acquires the wake lock, receives the results from the extBoard, and then executes the next tasks assigned to the smartphone. If next tasks in $T$ are assigned to the extBoard, the Task Controller sends a list of tasks to the extBoard, releases the wake lock, and switches to sleep state again. The Task Controller also continuously updates the meta information of tasks (e.g., execution times) as well as priorities of different branches.

Our measurement study shows that the smartphone consumes considerable energy during wake-up and tail phases (cf. Section 5.3). We optimize the design of Task Controller to minimize the number of smartphone wake-ups. After the Task Controller executes a task $T$ on the smartphone, it will check if there is any task in $\mathbb{G}$ assigned to the extBoard that takes $T$’s output as input. If yes, Task Controller will copy $T$’s output data to a memory area on the extBoard. This allows the extBoard 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 copied data may not be used by the extBoard eventually. However, typical signal processing pipelines likely contain a limited number of branches. DroidSense allows the developer to turn off this feature when the application has branches.

#### 6.4.2 Task Controller at extBoard

The Task Controller at the extBoard is an infinite loop that checks the arrival of messages from the smartphone as well as the status of a circular buffer that stores the most recent sensor samples. If it receives a start message from the smartphone, it creates a periodic timer. This timer fires every sampling period and generates a hardware interrupt. The timer interrupt handling routine reads a sensor sample from the ADC, time-stamps it, and then inserts it to the circular buffer. This process involves only a few instructions, and is optimized to reduce the interrupt handling delay. The infinite loop continuously checks if a signal segment has been filled in the circular buffer. If yes, the Task Controller initiates an execution cycle according to the execution tree $T$. Upon receiving a task execution request from the smartphone, it runs a list of tasks and activates the smartphone if remaining tasks need to be executed on the phone. Note that the timer interrupt handling routine can preempt the infinite loop at any
time, which ensures real-time sensor sampling.

7 Performance Evaluation

In this section, we first evaluate the overhead of a few DroidSense components. We then evaluate the effectiveness of our design using a case study, which comes from real sensing applications. As task partitioning is the key feature of DroidSense, we compare DroidSense’s Task Partitioner with three alternative designs: extBoard-only, phone-only, and greedy. The extBoard-only approach assigns all tasks to the extBoard. In contrast, the phone-only approach assigns all tasks to the smartphone. The greedy approach assigns the tasks to the extBoard in order until the processing delay upper bound $T$ is broken, and assigns the remaining tasks to the smartphone. Therefore, under the greedy approach, the smartphone wakes up and goes to sleep only once in each sampling duration. The drawback of this approach is that, if the application starts with a few computation-intensive tasks, the extBoard will be underutilized, leading to high smartphone energy consumption. The evaluation in this section uses three DroidSense nodes. The extBoard is Arduino Mega ADK [3]. However, we use three different Android smartphones as listed in Table 2. In this section, the DroidSense nodes equipped with LG GT540, Galaxy Nexus, and Nexus S are abbreviated as GT540, Galaxy, and Nexus, respectively.

7.1 Overhead of DroidSense

Overhead of JNI: As discussed in Section 6.1.1, all signal processing algorithms are implemented in C++ and accessed by the smartphone Task Controller via JNI. The first set of experiments measure the overhead caused by the JNI. Specifically, we measure the delays of invoking the native routine and returning from it. The overhead is calculated as the ratio of the total delay to the total execution time of the task. Fig. 11 plots the overhead of various tasks versus the length of input signal. We can see that the overhead decreases with the signal length. This is because the execution time of the algorithm becomes dominant for longer input signal. In particular, when the signal length is 1600 as specified for the case study in Section 7.2, the overhead is less than 8%. For the fine-grained wavefront arrival time picker [30], the overhead is about 30% if the signal length is 128. However, the total delay is about 70 microseconds, which does not cause significant delay. Nevertheless, as discussed in Section 6.1.1, by utilizing JNI, the burden in developing and maintaining two versions of each algorithm can be avoided.

Overhead of online partitioning: As discussed in Section 6.3.3, the Task Partitioner can be executed online periodically to be environment-adaptive. Fig. 12 shows the execution time 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 $n = 20$. As shown in the case study in Section 7.2, realistic applications often have fewer than 20 tasks. Therefore, the online task partitioning will not introduce significant overhead to the smartphone.

7.2 Case Study: Event Timing

In this paper, we use a case study to extensively evaluate the performance of DroidSense. This case study is an application that estimates the wavefront arrival time of a possible acoustic/seismic event, which is an important building block of many acoustic/seismic monitoring applications such as distributed event timing [27] and source localization. Fig. 13 shows the application specification of this case study. The incoming signal is first pre-processed by mean removal and bandpass filtering. Wavelet transform is then applied to the filtered signal. Signal sparsity and coarse arrival time are computed based on the wavelet coefficients. In the following, we present the evaluation results.

Effectiveness of Task Partitioner: We first evaluate the effectiveness of the task partitioning algorithm presented in Section 6.3.2, by comparing with the baselines. Fig. 14 shows the task partitioning results of various partition approaches. Note that the delay bound $D$ is set to 1.8s. The extBoard processing delay meets this bound except for the
Figure 14. The results of various partition schemes.

Figure 15. 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.

deExtBoard-only approach. Fig. 14(a) and Fig. 14(b) plot the estimated total energy consumption and total execution time (i.e., phone + extBoard) of a DroidSense 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 partition approach (“optimal”) achieves the lowest energy consumption on different smartphones.

We then evaluate the impact of the delay bound $D$ on the task assignment and smartphone energy consumption. The top part of Fig. 15 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 6.3.2. However, we can see a number of drops in the top part of Fig. 15. For instance, when $D$ increases from 1.37s to 1.38s, the number of extBoard tasks drops from 4 to 1. This is because a computation-intensive task replaces the previous four lightweight tasks to increase the CPU utilization of extBoard and reduce the smartphone energy consumption. The bottom part of Fig. 15 shows the smartphone energy consumption versus $D$. We can see that 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 DroidSense node to run the application over real-time sensor readings. Fig. 16(a) and 16(b) plot the measured extBoard processing delay and the smartphone energy consumption versus the specified delay bound. Note that the smartphone processing delay is less than 5 ms for all settings of delay bound. Therefore, the extBoard processing delay dominates. From Fig. 16(a) we can see that the specified delay bound is always met. Moreover, the extBoard processing delay increases with the delay bound, proving the effective utilization of the allowed extBoard CPU time. From Fig. 16(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: $1.2 \times 10^4$ mAh) versus duty cycle of extBoard under various settings of delay bound. The results are plotted in Fig. 17. When the duty cycle is 100%, the projected lifetime is 5.8 days, which well matches the result of 5.5 days in our field experiment in Section 8. When the duty cycle is 20%, the node can live for up to 2 months. As shown in Fig. 16(b), the smartphone energy consumption is tens of millijoules, while the extBoard energy consumption is about one joule if duty cycle is 100%. As the active powers of extBoard and smartphone are comparable (cf. Section 5), the extBoard energy consumption dominates if its duty cycle is large. In such cases, the majority role of the smartphone is to help meet the tight delay bound, and the node lifetimes are similar for different delay bounds. However, as shown in Fig. 17, when duty cycle is 20%, the lifetime can be extended by 18.4% if the delay bound increases from 0.1s to 2.0s. Nevertheless, with the help of smartphone, the DroidSense node can meet tight delay bounds, which is critical to the success of many sensing applications that requires continuous sensor sampling.

Effectiveness of data framing: The current version of signal processing library provides the meta records of five different input signal lengths for the four frameable tasks in Fig. 13, i.e., compute mean, remove mean, bandpass, and compute sparsity. Therefore, the framing parameter $K$ (cf. Section 6.3.4) can have 5 settings. We iterate among all the $5^4 = 625$ possible $K$ value combinations of the four tasks. For each combination, we run the Task Partitioner to obtain
a solution as well as the associated total energy consumption. We sort all solutions in terms of energy consumption in ascending order. Fig. 18 plots the energy consumption of the solutions at a few particular indexes, for different phones. We can see that by data framing, the most efficient solution leads to 1/9 energy consumption of the most inefficient one.

**Effect of branches:** To evaluate the effect of branches, we integrate the event timing application in Fig. 13 with an event detection approach [32]. Fig. 19 shows the block diagram of the application. The gray blocks are pre-processing algorithms; the white blocks are the earthquake detection algorithms [32]; the black blocks are the P-phase estimation algorithms [30].

The white blocks are pre-processing algorithms; the gray blocks are the earthquake detection algorithms [32]; the black blocks are the P-phase estimation algorithms [30].

8. **Deployment and Experiences**

In July 2012, six early-stage prototype DroidSense nodes shown in Fig. 1 were evaluated in a field experiment on Tungurahua Volcano, Ecuador. Fig. 21 shows the node locations and installation. Four of them were deployed for two days while the other nodes were deployed for a week. By the time of the deployment, we were still developing the Task Partitioner and Controller. Hence, each DroidSense node only ran a few preprocessing algorithms on the smartphone and stored the sensor data into the SD card. However, this deployment demonstrated the feasibility and advantages of DroidSense, and provided important lessons that guided our later design.

8.1 **Feasibility and Advantages of DroidSense**

We verified that there is cellular coverage on many areas of volcano, which could be leveraged for networking the distributed DroidSense nodes. This would allow a hybrid network architecture for a large-scale deployment: some nodes may serve as the cluster heads that transmit summaries of data to a remote server through the cellular network while other nodes communicate with their cluster heads through peer-to-peer links using short-range transceivers including WiFi (ad hoc mode and WiFi Direct) and Bluetooth.

We developed a GUI to draw the real-time sensor readings on the smartphone screen. This feature greatly eases node health testing in the field deployment. Moreover, we developed a node monitoring tool running on laptops, which uses ObexFS [14] to mount the SD card on the phone to a
Linux laptop over Bluetooth. We used this tool to check node status, downloaded and charted sensor data when revisiting the nodes during the deployment period. It also allowed us to configure the deployed nodes in the field. This feature avoided opening the enclosure and hence prevented heavy moisture from getting in and undermining the devices.

Each node was enclosed in an inexpensive waterproof case. The environment of the volcano is very humid. The first day of the deployment was raining. This proved a good case. The environment of the volcano is very humid. The moisture from getting in and undermining the devices.

8.2 Results and Learned Lessons
During the test period, the DroidSense nodes successfully captured all the earthquakes that were reported by a permanent volcano monitoring infrastructure. Fig. 22 plots the 3D seismic data recorded by a node during an earthquake. P-wave and S-wave can be easily differentiated, proving the quality of the sensor data. However, advanced volcano monitoring applications, such as hypocenter estimation, require sub-millisecond accurate and globally synchronized sensor sample timestamping. Our pre-deployment lab experiments showed that the on-phone GPS is insufficient to provide such a precision, because of the random delay in Android’s GPS access mechanism, which has a standard deviation of up to 32 milliseconds. We integrated an external GPS receiver with the IOIO extBoard. However, due to the limitations of IOIO firmware, the microsecond-precision 1-pulse-per-second (1PPS) hardware interrupt from the GPS receiver cannot be accurately captured by IOIO. As a result, the timestamping across multiple nodes are not well synchronized and the hypocenters of the detected earthquakes cannot be accurately estimated. After this field experiment, we replaced IOIO with Arduino, which is able to precisely capture the 1PPS signal. Our current DroidSense nodes can achieve high-quality real-time sampling, microsecond-precision GPS synchronization and timestamping.

9 Conclusion and Future Work
This paper presents DroidSense, a toolkit for smartphone-based data-intensive sensing systems. DroidSense features a two-tier hardware architecture and provides a rich and extensible library of signal processing algorithms. It dispatches the execution of the signal processing algorithms either to the smartphone or to the extension board to minimize energy consumption subject to bounded processing delay. In the future, we will extend DroidSense to support distributed signal processing and collaborative data analysis across multiple nodes, with the consideration of wireless communication power consumption in the design of Task Partitioner.

10 References