Smartsensors: Energy-efficient sensing on mobile devices

Silviu Jingoi
University of Toronto

Wilson To
University of Toronto

Italo De Moraes Garcia
University of Toronto

Kevin Lee
University of Toronto

Eyal de Lara
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Applications that perform continuous sensing on mobile phones have the potential to revolutionize everyday life. Examples range from medical and health monitoring applications, such as pedometers and fall detectors, to participatory sensing applications, such as pollution and traffic monitoring. Unfortunately, current mobile devices are a poor match for continuous sensing applications as they require the device to remain awake for extended periods of time, causing fast battery drainage. This paper presents *Smartsensors*, a new approach that offloads the sensing portion of the continuous sensing application to a low power processor and wakes up the main processor only when an event of interest occurs. Smartsensors differ from other heterogeneous architectures in that developers are presented with a programming interface that lets them define application specific wake up conditions by selecting among a set of predefined filtering algorithms and tuning their parameters. Smartsensors achieve performance that is comparable to approaches that provide fully programmable offloading, but do so with a much simpler programming interface that simplifies deployment and portability.

1 Introduction

Today smartphones and tablets are used primarily to run interactive foreground applications, such as games and web browsers. As a result, current mobile devices are optimized for a use case where applications are used intermittently during the day, in sessions that last for several minutes. For example, Alice takes a little break to pick up her phone, check the weather, read the latest news story and play a game for a few minutes, and then puts it away to go back to work. To maximize battery life, current mobile devices are engineered to go into sleep state for most of the day when they are not supporting such interactive usage.

Unfortunately, most mobile platforms are a poor match for a growing class of mobile applications that perform continuous background sensing. Examples range from context-aware applications [6, 11], such as targeted advertising, and medical applications [10, 23, 29] that improve our well-being or even save lives using activity recognition (e.g., fall detection), to applications that use participatory sensing to get a better understanding of the physical world, such as noise pollution [17, 18] monitoring or traffic prediction [12]. While the processing demands of these applications are modest most of the time, they require periodic collection of sensor readings, which prevents the device from going to sleep for extended periods of time. As a result, applications that perform continuous sensing may cause the device's battery to drain within several hours.

To improve support for continuous sensing applications the research community has proposed the use of fully programmable heterogeneous architectures [15, 24, 28]. In these approaches, developers partition their applications to offload the initial filtering stages of the application to a low-power processor (or a hierarchy of processors). When the code running on the low power processor detects the occurrence of event of interest, it proceeds to wake up the phone and passes control to the rest of the application. While the ability to run custom code on the low-power hardware provides great flexibility, the significant complexity inherent in this approach has so far prevented its adoption in commercial devices. Instead, smartphone manufacturers, realizing the potential of sensing applications, have recently incorporated low-power processors into their architectures, but have limited application developers to APIs that provide fixed functionality, by either batching sensor readings or recognizing a small number of predefined activities that can be used as wake-up conditions [1, 3, 4].

In this paper we argue that these APIs are insufficient for supporting a rich and flexible set of continuous sensing applications. Batching is inefficient for applications

that depend on infrequent events and is not appropriate for applications that require crisp response time. An activity recognition API provides little flexibility with no support for applications interested in events that are not covered by the set of predefined activities.

Instead, we argue that the sensing API should expose access to filter-level functionality. We present *Smartsensors*, an approach that enables programmers to create custom wake-up conditions by choosing among a set of predefined filtering algorithms and tuning their parameters. By providing developer with access to lower-level filters that implement commonly used algorithms (e.g., FFT, exponential average), as opposed to higher-level activity detectors, Smartsensors provide a better balance between flexibility and ease of deployment. Smartsensors allows developers to create a broad set of wake-up conditions that can be used to detect a wide range of activities. Simultaneously, they are significantly easier to program compared to fully programmable offloading. Moreover, since the filters are pre-specified, their implementations can be optimized (by the device manufacturer) for each low-power processor, improving application portability between devices. Furthermore, it is possible to combine the filtering and wake-up operations when applications register interest in the same set of events. While Smartsensors could be programmed directly by individual applications, it is easy to imagine user-level activity recognition libraries that encapsulate the functionality of the Smartsensor to provide simple wake-up conditions for a large number of activities.

To evaluate the benefits of Smartsensors, we built a prototype implementation that extends a Nexus 4 phone with a low-power sensor board. To enable us to conduct controlled and repeatable experiments, we mounted the Smartsensor prototype on an AIBO ERA 210 robot dog, and we developed three applications that use accelerometer readings to detect actions that the robot performs (e.g., walking, standing and seating). Experiments conducted on traces collected with this setup, as well as limited traces collected from human subjects, show that Smartsensors can reduce the average energy required to run continuous sensing application by up to 94%, while matching the detection recall and precision of an approach that keeps the phone awake at all times. Moreover, for most of our usage scenarios,the Smartsensor approach achieves over 95% of the potential power saving, indicating that an implementation that supports custom code offloading will achieve only marginal additional improvements.

The rest of this paper is organized as follows. Section 2 discusses existing approaches to continuous sensing and their limitations. Section 3 introduces our new approach for continuous sensing, which lets developers configure the filters used to implement wake-up conditions. Section 4 describes our Nexus 4-based prototype. Sections 5 and 6 present our experimental setup and the results from our evaluation. Finally, Sections 7 and 8 describe our work in the context of related work and conclude the paper.

2 Background

Mobile application developers can choose among several existing approaches when attempting to perform continuous sensing. The availability of some of these approaches depends on the mobile device's hardware and operating system.

2.1 Always Awake

The basic approach for continuous sensing is to keep the mobile device *always on*. As an example, on the Android operating system, the sensing application acquires a wake lock to prevent the device from sleeping and registers a listener for a specific sensor. Whenever the sensor produces a new reading, the operating system passes the reading to the application via a callback function. This operation continues until the listener is unregistered or until the wake lock is released. The benefit of this approach is that it achieves the highest possible accuracy because all the sensor data is delivered to the application. However, the device cannot a sleeping state and hence the power consumption of this approach is high.

2.2 Duty Cycling

To avoid keeping the device awake for extended periods of time, *duty cycling* allows a device to sleep for fixed, usually regular, periods of time. An application on Android can implement duty cycling by scheduling a wake-up alarm using the system's alarm service and by ensuring that it has not acquired a wake lock. When no application has a wake lock, the operating system can power down the processor. When the wake-up alarm fires, the processor wakes up and notifies the application. Next, the application collects sensor readings for a short period of time, and then reschedules another alarm.

Duty cycling has several drawbacks. While the device is sleeping, events of interest will not be detected because the application does not receive any sensor data that were produced at those times. Additionally, during the short time periods when the device is awake and collecting sensor readings, the events of interest might not occur. These power transitions are wasteful and expensive (as we show later).

The basic duty cycling approach can be improved by dynamically adjusting the wake-up time. When an application is woken up and detects an event of interest,

it continues to collect sensor readings until events do not occur for some period, after which it returns to its regular duty cycling schedule. This improves detection accuracy (e.g., event recall) but at the expense of increased energy consumption. Unfortunately, finding a good balance between accuracy and power savings depends on the specific application and the user's behaviour. Another drawback of this approach is that is does not scale well with multiple applications. It is possible for mutually unaware applications to implement conflicting wake-up schedules, resulting in little or no sleep.

2.3 Sensor Data Batching

An extension to duty cycling is *sensor data batching* in which the device hardware is able to collect sensor readings while the main processor is asleep. Upon wake-up, the entire batch of sensor data is delivered to the application(s) that registered a listener for that specific sensor. Unlike duty cycling, batching requires hardware support (e.g., it is currently only available on the newly released Google Nexus 5, running Android 4.4), but it enables applications to receive all the sensor data, and hence provides detection accuracy similar to the Always Awake approach. However, batching affects detection timeliness. Applications may detect events of interest many seconds after they actually occurred. Also, as with the duty cycling approach, the device may wake up to find out that no events of interest have occurred in the current batch, which is wasteful.

2.4 Computation Offloading

Generalizing batching, a low-power peripheral processor can be used to collect and *compute* on sensor data while the main processor is asleep [15, 28]. This approach allows developers to offload their own sensor data analysis algorithms to the peripheral processor, enabling arbitrarily rich sensing applications. However, developers are exposed to the full complexity of the architecture, making application development more difficult. Even simple sensor-driven applications have to be refactored into distributed programs, and the code that is offloaded has to be chosen carefully so that it can run in real time. The latter is complicated because it depends on the type and functionality of the peripheral processor that is available. As a result, multiple versions of the application may be required to handle hardware variations across phones. Furthermore, since each application is written independently, this approach makes it hard for multiple applications to program or use the same sensor.

2.5 Activity Detection

Given the challenges with full computation offloading, recent mobile devices limit offloading to a small, predefined set of *activity detection* algorithms. For example, the Motorola Moto X [3] uses two low-power peripheral processors to wake up the device when certain events occur. A dedicated natural language processor is used to wake up the device when a user says the phrase "OK Google Now". Also, a contextual processor is used to turn on a part of the screen when the mobile device is taken out of a pocket, or start the camera application when the processor detects two wrist twists. Additionally, the latest Android 4.4 [1] supports specific wake-up events based on detecting "significant motion" or a step, if a device has appropriate sensors. However, the application developer has no control over the underlying activity detection algorithms, including customization of parameters. This approach is easy to use, and it provides good energy savings because the device only wakes up when the event of interest occurs. However, it only works for applications that can take advantage of the predefined set of activities, limiting the number and types of applications that can be supported.

3 Smartsensors

Smartsensors make possible continuous mobile sensing by providing a filter-based sensor API for application programming. Applications create *custom wake up conditions* for events of interest by choosing from a prespecified set of filtering algorithms and tuning their parameters. When these events occur, the main processor is woken up and the application code is invoked. The result is that applications view the sensors as "smart" sensors that generate relevant events only. This approach enables supporting a large number of sensing applications while being significantly easier to program compared to fully programmable offloading. Since the filters are prespecified, the device manufacturer can optimize their implementations for each low-power processor, improving application portability between devices. Furthermore, it is possible to optimize the operation of multiple applications that use common filters.

Figure 1 shows the architecture of a system that implements the Smartsensors abstraction. Applications interact with a sensor manager and define custom wake-up conditions by choosing among the available pre-defined filters. The figure also shows that the architecture supports recognition libraries that encapsulate the functionality of the Smartsensor to provide simple wake up conditions for a large number of activities. To keep the complexity of filters low, our initial implementation limits filters to operations on data collected from a single sen-

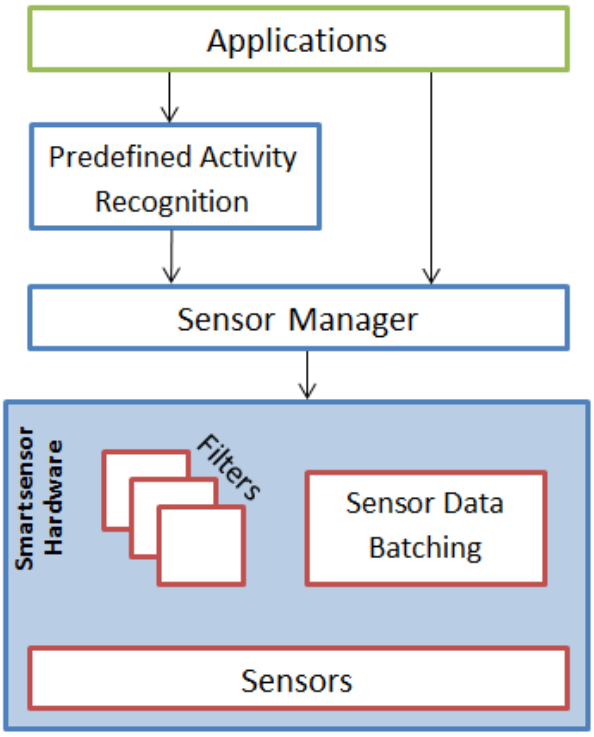

Figure 1: Smartsensor system architecture.

sor. Applications that perform sensor fusion, are implemented by defining separate independent wake up conditions on multiple sensors, and merging data on the main processor.

The main challenge with our Smartsensors approach is defining the appropriate set of event filters for each sensor. First, there is a trade-off between filter generality and accuracy. Simple generic filters can support a large set of applications, albeit no specific application is likely to experience optimal performance. Conversely, a highly specialized filter may provide optimal performance but is only applicable to a limited set of applications. Second, there is also a trade-off between filter complexity and power savings. More complex filters can reduce energy consumption by preventing unnecessary wake-ups due to increase accuracy. On the other hand, more complex filters have higher computational demands, which require a larger and hungrier peripheral processor.

Fortunately, continuous sensing applications are commonly structured as modular pipelines of increasing selectivity and computational complexity. Early stages in the pipeline tend to be simpler but execute with much higher frequency as only data that passes the selection criteria is forwarded along to the next stage. Moreover, while the application logic of different sensing application varies widely, applications that use the same sensors tend to make use of similar algorithms in the early stages

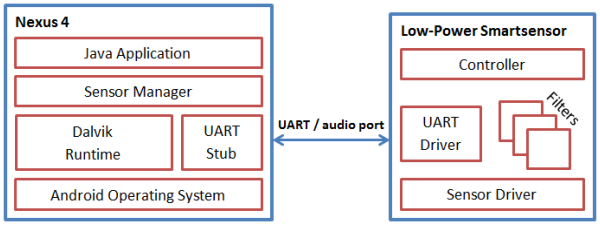

Figure 2: Prototype architecture.

of their event detection pipelines. We can thus determine potential filter candidates by identifying these common algorithms.

For example, several accelerometer-based applications initially check whether the acceleration magnitude exceeds a certain threshold in order to determine if there is any movement. Also, the acceleration data has significant noise, and hence many applications implement low-pass filters to smoothen the accelerometer readings [13, 14]. If multiple accelerometer-based applications are running at the same time, they are each checking for these events independently, possibly resulting in redundant computation. Common steps such as thresholding and low-pass filtering are excellent candidates for event filters. Similarly, audio data is commonly passed though a high-cut or a low-cut filter in order to reduce the amount of treble or bass in the sound. Image processing makes use of low-pass, high-pass and band-pass filtering for noise reduction, and edge detection [20, 19].

4 Prototype

We implemented a prototype of our system using a Google Nexus 4 phone running Android 4.2.2. The low-power Smartsensor is implemented using a Texas Instruments micro-controller board attached to an accelerometer sensor. We chose to focus our efforts on the accelerometer sensor because of its relative simplicity and the availability of a wide range of accelerometer-based applications on various mobile application markets. Figure 2 shows a high-level diagram of the prototype's architecture.

The Nexus 4 and TI board communicate over the UART port made available by the Nexus 4 debugging interface via the physical port used by the audio interface. The serial connection provides sufficient bandwidth to support low bit-rate sensors, such as the accelerometer, a microphone or GPS. However, extending the prototype to work with higher bit-rate sensors like the camera would require a higher bandwidth data bus, such as I^2C.

The rest of this section describes our extensions to the phone, the implementation of the Smartsensor board, and the applications and test frameworks we use in our exper-

iments.

4.1 Nexus 4

On the Nexus phone we extended Android's SensorManager to include the new features made available by our system. To prevent a steep learning curve, our goal was to provide an API very similar to Android's existing sensors API.

We extended the Sensor Manager to allow users to define custom wake-up conditions by specifying a predefined data filter and configuring the filter and wake-up parameters. The available types of sensor data filters, described in detail in Subsection 4.2.1, are a set of predefined implementations of a DataFilterAlgorithm interface. Figures 3 and 4 show an example of an application that registers a listener to receive accelerometer readings using the default Android Sensor Manager, and the modified code that includes a wake-up condition that uses an exponential moving average low-pass filter and an x-axis threshold, respectively.

We also created a UART stub to facilitate communication between the mobile phone and the Smartsensor board. The UART stub is called when the Smartsensor board detects an event based on its wake-up condition. In turn, the UART stub is a shell script that notifies the SensorManager using an Android Intent[2].

For our prototype we avoid modifying the Android kernel in any way. Because the current prototype uses the UART port for communication with the Smartsensor, we were able to constrain our implementation to the user space. An integrated implementation would likely make use of a higher bandwidth bus such as the I^2C bus and require a custom driver.

We power profiled the Google Nexus 4. The results are summarized in Table 1. During all the measurements, the device's screen, WiFi and GPS were turned off. While the device is sleeping, its power usage is very low, consuming only 9.7 mW. While awake, the power consumption is significantly higher, averaging 323 mW. During our power measurements we noticed that additional energy is consumed during transitions between the asleep and awake states. Each transition takes about 1 second. During a wake-up transition, the average power consumption goes up to 384 mW, while during an awake-to-asleep transition the average power consumption is 341 mW.

4.2 Smartsensor Board

The low-power Smartsensor board consists of a driver that talks to the accelerometer sensor, a set of filtering algorithms, a controller, and an UART driver for communicating with the phone. The controller orchestrates the execution of the filters, and uses the UART driver to wake up the phone when a wake-up condition is satisfied. The UART driver opens a serial port and then uses a TTY to copy the sensor readings and start the shell script that notifies the Sensor Manager.

4.2.1 Sensor Data Filters

We implemented three filters of increasing complexity:

Null This algorithm performs no filtering. It simply forwards the raw accelerometer readings.

EMA Applies an exponential moving average low-pass filter that removes some of the noise from the sensor data and is computationally cheap, requiring only a few algebraic operations for every sensor reading. The filter takes an alpha parameter between 0 and 1, that controls the "smoothness" of the filtered data, and implements the following equation: $out_t = (1-\alpha) \times out_{t-1} + \alpha \times in$

FFT Applies a low-pass filter based on Fast Fourier transformations [14] that is more accurate at removing noise from the sensor data. However, it is computationally expensive. Two parameters can be set for the FFT: a window-size and a relative energy threshold value. The window size indicates the number of readings that are to be used in each discrete Fourier transform. The relative energy threshold value controls the amount of energy in the output data. A lower relative energy threshold value would result in "smoother" output sensor values.

4.2.2 Wake-up conditions

Programmers specify a wake-up condition by selecting a filtering algorithm and setting one or more thresholds. The prototype supports the following thresholds:

Minimum Threshold is satisfied when the acceleration exceeds the threshold value

Maximum Threshold is satisfied when the acceleration goes below the threshold value

Threshold Range is satisfied when the acceleration is between two threshold values

4.2.3 Hardware Options

We evaluated two low-power micro-controllers manufactured by Texas Instruments, the MSP430 and the Stellaris LM4F120H5QR. The MSP430 has the advantage of requiring very little power, consuming only 3.6 mW while

```
SensorManager mSensorMan = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
Sensor mAccelerometer = mSensorMan.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
mSensorMan.registerListener(new MySensorEventListener(), mAccelerometer);
```

Figure 3: Typical usage of Android's SensorManager

```
SensorManager mSensorMan = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
Sensor mAccelerometer = mSensorMan.getSmartSensor(Sensor.TYPE_ACCELEROMETER);
DataFilterAlgorithm mDataFilter = new ExponentialMovingAverageLowPassFilter(0.75);
WakeUpCondition mWakeUpCond = new MinThresholdWakeUpCondition(mDataFilter, Sensor.AXIS_X, 12.0);
mSensorMan.registerListener(new MySensorEventListener(), mAccelerometer, mWakeUpCond);
```

Figure 4: Usage of the SensorManager with a wake-up condition.

Figure 5: AIBO ERS 210 with Smartsensor prototype.

awake. However, it has limited memory and cannot perform complex analysis of sensor data in real-time. In our tests, it was unable to run the FFT-based low-pass filter in real-time. The Stellaris LM4F120H5QR is powered by a Cortex-M4 processor. It can batch a higher number of accelerometer readings and can run all our filters in real-time. However, this micro-controller has an energy footprint an order of magnitude greater than the MSP430, consuming an average of 49.4 mW while awake.

4.3 Test Platform and Applications

To enable us to conduct controlled and repeatable experiments, we mounted the Smartsensor prototype on the back of an AIBO ERA 210 robot dog (see Figure 5). Because the robot's actions can be scripted, this setup provides an efficient and reliable way to determine ground truth. In contrast, labelling data collected from human subjects with ground truth is error prone and labour intensive.

We developed three applications that detect activities that the robot can perform: walking, sitting and standing, headbutts. We chose these actions because they have similar acceleration signatures to human activities. A walking robot has a similar acceleration signature as its human counterpart, though at a lower intensity. The headbutts are meant to represent very infrequent human actions such as falling. We found that robot stance transitions between the normal and sitting postures are very similar in their acceleration signature to humans sitting down and standing up. In Section 6 we show that the energy saving measured in our experiments with the robot approximate closely the results of experiments conducted on limited traces collected from human subjects.

Steps Counts how many steps the robot takes when it walks. It's algorithm is based on the human step detection algorithm proposed by Ryan Libby in [14]. The application takes in raw accelerometer readings and applies a low-pass filter on the x-axis acceleration. It then searches for local maxima in the filtered x-axis acceleration. Local maxima between $2.5\,m/s^2$ and $4.5\,m/s^2$ are detected as steps, given that no other steps were detected within the last 100 ms.

Transitions Detects transitions between sitting and standing. The application monitors changes in acceleration due to gravity on the y and z axes to determine the orientation of the device. If the z-axis (up-down relative to the dog) acceleration is between $9m/s^2$ and $11m/s^2$, and the acceleration on the y-axis (front-back relative to the dog) is between $-1m/s^2$ and $1m/s^2$, the device is in a horizontal position and the robot is assumed to be in a standing posture. Similarly, if the z-axis acceleration is

State	Average Power Consumption (mW)	Average Duration
Awake, running a pedometer application with data from the internal accelerometer	323	N/A
Asleep	9.7	N/A
Asleep-to-Awake Transition	384	1 second
Awake-to-Asleep Transition	341	1 second

Table 1: Google Nexus 4 power profile.

between $7.5 m/s^2$ and $9.5 m/s^2$, and the acceleration on the y-axis is between $3.5 m/s^2$ and $5.5 m/s^2$, the device is in an angled position and the robot is assumed to be in a sitting posture. The application detects transitions by looking for posture changes.

Headbutts Detects a sudden forward head movement. The application monitors the y-axis acceleration and searches for local minima between $-3.75\, m/s^2$ and $-6.75\, m/s^2$.

5 Experimental Setup

We evaluated our implementation using trace-based simulation on traces collected from controlled experiments conducted with our robotic testbed as well as limited traces collected from human subjects. While our robotic testbed allows us to run live experiments, we chose instead to use trace-based simulation for several reasons. First, it took the robot close to an hour to complete a single experiment. A thorough exploration of the configuration space of the various sensing approaches we consider would have required over a year of continuous live experiments. Moreover, taking fine grain power consumption measurements while the robot is in motion is not trivial.

5.1 Trace Collection

We collected our traces by having the robot perform multiple runs with a prototype smartphone attached to it back. The smartphone ran an application that kept the device always awake and continuously recorded accelerometer readings for all three axes. Each run generated a trace that included timestamps for the start and end of each action performed by the robot (which we use as the ground truth for our experiments) and a list of timestamped acceleration readings.

In each run, the robot performed five different actions: standing idle, walking, sit-to-stand transitions, stand-to-sit transitions, and headbutt. We created runs with three different levels of activity. Runs in groups 1, 2 and 3 spent 90% , 50% and 10% of the time standing idle, respectively. The reminder of the time was allocated as follows: 73% for walking, 24% for transitions between sitting and standing, and 3% for headbutts. This setup allows us to experiment with detecting actions that are common, somewhat frequent, and rare. In total, the robot executed 18 different runs: 9 for the group 1, 6 for group 2 and 3 for group 3. We generated more runs for groups 1 and 2 because of the lower activity levels compared to group 3. To eliminate bias, the list of actions was generated randomly for each run, based on the expected probabilities of each action occurring.

In addition, to validate the applicability of our results to human scenarios, we collected six hours worth of accelerometer traces from three different individuals while they perform routine daily activities: morning commute using public transit, walking in a retail store, and walking in an office. Between 20% and 37% of each trace is spent walking.

5.2 Sensing Approaches

We replayed each trace under the following sensing approaches:

Always Awake The applications keep the phone awake all the time, constantly collecting accelerometer readings. This setup achieves the highest detection recall[1] and precision[2] and provides a baseline for comparison.

Duty Cycling We modified the applications so that they check sensor readings periodically and then put the phone to sleep. On wake-up, the phone is kept awake for 4 seconds in order to collect sensor data. If an action is detected, the phone is kept awake for another 4 second, and goes to sleep otherwise. This software only implementation runs on any mobile device and does not require special hardware support.

Batching This configuration emulates hardware support for collecting and batching accelerometer readings while the main processor sleeps (e.g., Nexus 5). The application wakes up periodically, reads the batch of sensor readings, runs the detection algorithm, and goes back to sleep.

Predefined Activity This configuration emulates hardware support for recognizing a limited number of predefined activities (e.g., Moto X). For this purpose,

[1] $Recall = TruePositives/(TruePositives + FalseNegatives)$
[2] $Precision = TruePositives/(TruePositives + FalsePositives)$

Action	*Data Filter*	*Threshold(s)*
Steps	*EMA (alpha = 50%)*	*x-axis* $\geq 2m/s^2$
Headbutts	*FFT (relative energy threshold = 10%)*	*y-axis* $\leq -3.5m/s^2$
Transitions	*EMA (alpha = 10%)*	$2.25m/s^2 \leq$ *y-axis* $\leq 3.25m/s^2$ $8.2m/s^2 \leq$ *z-axis* $\leq 9.2m/s^2$

Table 2: Best performing Smartsensor wake-up conditions.

we hard-coded the Smartsensor board as a *step detector* (EMA filter with $alpha = 50\%$ and x-axis $\geq 2m/s^2$). The applications register an even handler for the *step activity* and put the phone to sleep. On wake-up, the applications acquire a batch of sensor readings, run their detection algorithm, and go back to sleep.

Smartsensors We let applications define their own custom wake-up condition by selecting a filter among the predefined set described in Section 4 and setting the filter's configuration parameters and thresholds . The applications register an even handler for the custom event of interest and put the phone to sleep. On wake-up, the applications acquire a batch of sensor readings from the Smartsensor, run their detection algorithm, and go back to sleep. Table 2 shows the wake-up condition for each application that achieves the lowest power consumption while matching the detection recall of the Always Awake approach.

Oracle A hypothetical ideal implementation that only wakes up when an event of interest occurs. Such a wake-up condition would achieve the same detection precision and recall as Always Awake, with the lowest possible energy consumption. The difference between the power consumption of this method and the Smartsensors approach provides an upper bound on the potential additional benefits of custom code offloading.

For each sensing approach and trace, we measured the amount of sleep and awake time, the total number of wake-up events, and the recall and precision of the application. Finally, we used an energy model derived from measurements of our Smartsensor prototype to estimate the average power consumption. For experiments with *Always Awake* and *Duty Cycling*, the power model accounts only for the energy consumption of the Nexus 4. For *Batching* and *Predefined Activity*, the model also includes the cost of the simple TI MSP430 microcontroller. Finally, experiments that use *Smartsensors* and *Oracle* include the cost of either the TI MSP430 or the TI Stellaris LM4F120H5QR, depending on the application being evaluated.

6 Results

In this section we first present the result of experiments conducted on synthetic traces collected with our robotic testbed. We then present results from experiments on a small number of traces collected from human subjects.

6.1 Synthetic Traces

We answer the following questions:

1. What are the benefits of Smartsensor over Duty Cycling and Batching?
2. How flexible is Predefined Activity?
3. How much additional benefit can be obtained from fully programmable wake-up conditions?
4. Are multiple different filters required, and how important it is to let the application configure the filter parameters and thresholds?

6.1.1 Smartsensor vs. Duty Cycling and Batching

Table 3 presents the results of replaying the traces collected from the robot for the sensing approaches described in the previous section. For each application, the table presents average power consumption and activity recall. Results are averages across runs of the same group. We use the results for the Always Awake approach (which achieves close to perfect recall) as a base line for comparison. All sensing approaches achieved similar average precision (Headbutts: 89%, Transitions: 91%, Walking: 0.93%), and we therefore do not include these numbers in the table.

As expected, Duty Cycling performs poorly. Short sleep intervals actually result in an increase in power consumption due to frequent transitioning between awake and asleep states. Longer sleep interval are more effective at saving energy, but they do so by sacrificing recall. For example, a sleep interval of 10 seconds reduces the Headbutts and Transitions recall bellow 30%.

Batching matches the recall of Always Awake, but requires long batching intervals to achieve large energy savings. Therefore, this approach is not be appropriate for applications with timeliness constrains. For example, the user of a gesture recognition application [16, 26] would not be satisfied if the application detects the performed gesture after a delay of more than a couple of seconds. We anticipate that in practice realistic batching intervals are in the order of a few seconds. We therefore conclude, that batching will result in significant energy

	Approach	Sleep Interval (seconds)	Average Power (mW)			Recall		
			Steps	Headbutts	Transitions	Steps	Headbutts	Transitions
Group 1 90% idle	Always Awake		323			98%	100%	100%
	Batching	2	350					
		5	186					
		10	115					
		20	80.1					
		30	68.3					
	Duty Cycling	2	339			94%	57%	97%
		5	217			82%	14%	47%
		10	140			63%	29%	28%
		20	86.3			48%	14%	32%
		30	63.1			31%	7%	12%
	Predifined Activity		48.3			98%	36%	87%
	Smartsensor		48.3	60.3	18.6		100%	100%
	Oracle		41.5	59.6	16.4			
Group 2 50% idle	Always Awake		323			99%	100%	100%
	Batching	2	350					
		5	186					
		10	115					
		20	80.1					
		30	68.3					
	Duty Cycling	2	334			92%	74%	90%
		5	243			79%	31%	53%
		10	176			65%	20%	36%
		20	106			33%	5%	10%
		30	80			25%	11%	12%
	Predifined Activity		188			99%	13%	73%
	Smartsensor		188	65.1	43.3		100%	100%
	Oracle		153	62.3	29.5			
Group 3 10% idle	Always Awake		323			97%	100%	100%
	Batching	2	350					
		5	186					
		10	115					
		20	80.1					
		30	68.3					
	Duty Cycling	2	332			89%	47%	89%
		5	257			76%	28%	26%
		10	198			64%	27%	25%
		20	133			42%	19%	19%
		30	99.1			30%	9%	14%
	Predifined Activity		321			97%	95%	86%
	Smartsensor		321	65.7	51.7		100%	100%
	Oracle		266	62.9	34.9			

Table 3: Event recall and average power for synthetic traces.

Filter	Power Consumption (mW)		
	Steps	Headbutts	Transitions
Null	191	186	67.8
EMA	**188**	228	**43.3**
FFT	217	**65.1**	88.7

Table 4: Different wake-up conditions.

waste for applications interested in low frequency events (e.g., gesture recognition, fall detection).

Smartsensor matches the recall of Always Awake while achieving large reductions in average power in all scenarios other than step detection in Group 3, where walking represents 63% of the trace and the device experiences little sleep. Smartsensor performed best when the event of interest occurs infrequently, reductions power consumption by up to 94%.

6.1.2 Smartsensor vs Predefined Activity

Predefined Activity performs very well for step detection (as expected), but results in either poor recall or low energy savings for the other two applications. We conclude that hardware support for predefined activity detection may work well for specific use cases, but lacks the generality to support a broad set of applications.

6.1.3 Smartsensor vs. Oracle

By comparing the performance of the Smartsensor approach to Oracle we observe that for most usage scenarios, the Smartsensor approach achieves over 95% of the available power saving[3]. We conclude that a system that allows custom code offloading to the low-power processor can achieve minimal additional power savings.

6.1.4 Filter Selection and Configuration

Table 4 shows the average power for wake-up conditions using different filters for runs in Group 2. For each wake-up condition, the filter uses the parameters and thresholds that minimize average power while matching the recall of Always Awake. The condition that achieves the lowest average power is highlighted. The results show that different filters are optimal for different applications. Wake-up conditions based on EMA performed better than those based on FFT for Steps and Transitions. While the FFT filter reduces the occurrence of unnecessary wake-ups (cases where the main processor is waken up but does not find an event of interest), this benefit it overshadowed by the additional power required to run it.

Even in cases where the same filter type is optimal, the configuration parameters for the filter and the wake-up thresholds differ significantly (see Table 2). Figure 6 uses the Headbutts application to illustrate the importance of threshold selection. A threshold that is too strict causes a significant drop-off in the achieved recall. However, a threshold that is too lenient results in additional power consumption without any extra benefit to recall because of unnecessary wake-ups.

[3] $AvailableSavings = 1 - Oracle/AlwayAwake$

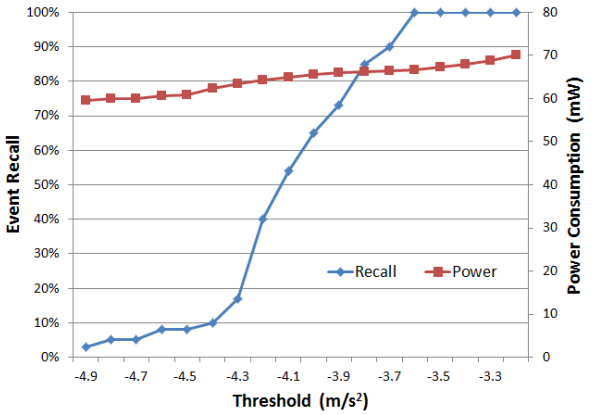

Figure 6: Threshold sensitivity for Headbutts.

6.2 Human Traces

Table 5 shows the results from running the step detector application on traces collected from three human subjects. Since these traces are not annotated with ground truth, we use the steps detected by the Always Awake as the baseline for determining recall.

Remarkably the results from these experiments show very similar benefits to the synthetic experiments for runs with low and medium levels of activity. For example, the Smartsensor approach achieves at least 91% of the available power saving in each of the traces.

7 Related Work

The idea of waking up a device when an event of interest occurs has been around since the inception of mobile phones. The phone's radio transceiver wakes up the device when an incoming call or a text message is received [5]. Wake on Wireless [27] extended this idea by augmenting a PDA with a low-power radio that would send a wake-up message when an incoming call is received. Similarly, Wake on WLAN [21] allows remote wake-up of wireless networking equipment.

Turducken [28] generalizes the "wake on event of interest" approach to several types of applications and to multiple components operating at increasingly small power-levels. Little Rock [24] applies Turducken's multi-tiered architecture to sensing on mobile devices.

Approach	Sleep Interval (seconds)	Average Power (mW)			Average Recall
		Trace 1 37% walking	Trace 2 22% walking	Trace 3 20% walking	
Always Awake		323			100%
Duty Cycling	2	329	330	330	97%
	5	272	260	261	92%
	10	220	195	198	82%
	20	172	131	134	66%
	30	148	104	106	57%
Batching	2	350			100%
	5	186			100%
	10	115			100%
	20	80.1			100%
	30	68.3			100%
Smartsensor		136	77.9	72.6	100%
Oracle		117.8	65.8	62.3	100%

Table 5: Event recall and average power for human traces.

Reflex [15] complements the idea proposed by Turducken by providing a shared memory abstraction to be used by the different processors.

Smartsensors differs from these approaches by hiding the heterogeneous nature of the system from the application developer. Creating an application that makes use of one or more Smartsensors does not require the developer to create any code that will run on the low-power processor(s). We limit the available interface in order to increase portability, while still achieving the majority of the potential power savings.

Smartphone manufacturers have started to incorporate low-power processors into their architectures, but have only implemented limited APIs that provide fixed functionality. The Google Nexus 5 allows batching of sensor readings [1, 4], and the Motorola Moto X provides recognition for a small number of predefined activities that can be used as wake-up conditions [3]. While these wake-up conditions work well for some applications, they are inefficient for many other types of applications that are not interested in the set of predefined activities.

Most of the previously noted works focused on system architecture modification in order to lower the cost of sensing. Alternative approaches have also been explored. Ace [22] is a middleware that supports continuous context-aware applications while mitigating sensing cost for acquisition of context attributes (such as AtHome and IsDriving). It achieves power savings when multiple applications request strongly correlated context attributes. Additionally, it can reduce power consumption when a "cheaper" sensor exists, which can determine the value of a different context attribute that has a strong correlation with the requested context attribute (e.g. use the accelerometer to check if the user is jogging instead of using the GPS to determine if the user is at work). A middleware such as Ace is a great example of a library that can run on top of a Smartsensor architecture (once we implement support for multiple sensors) and achieve additional power savings. Sensor fusion has also been an active focus of related research. Data from multiple sensors can be used to increase context-awareness in mobile devices [9, 7].

While our focus was on power-efficient acquisition of sensor data, next generation mobile perception applications face related problems regarding partitioning of application code. MAUI [8] enables fine- grained energy-aware offload of mobile application code to remote servers. Similarly, Odessa [25] uses code-offloading to address the issue of processing excessive amounts of sensor data on resource constrained mobile devices.

8 Conclusion

In this paper we proposed Smartsensors, an approach that uses a low-power processor to perform sensor data acquisition and wakes up the main processor when an event of interest occurs. The API provided to developers hides the heterogeneous nature of the system, ensuring application development does not increase in complexity. The use of an API also allows Smartsensor-based mobile applications to be portable to other Smartsensor-enabled devices. Our experiments showed that the use of SmartSensors can match the detection recall of an approach that keep the phone constantly awake and reduce average power consumption by up to 94%. In most usage scenarios we explored, we were able to achieve over 95% of the power savings that would be achieved by a "perfect" wake-up condition. This suggests that a system that allows custom code offloading to the low-power processor can achieve minimal additional power savings.

Our immediate future work includes developing

Smartsensors based on other sensors available on mobile devices, such as the microphone, camera or location sensor. We believe application developers may face challenges in selecting the optimal data filter and wake-up parameters for their application. We plan on creating self-learning algorithms that, given feedback from the application about events of interest, will be able to determine the optimal data filter and wake-up parameters. Additionally, we will be exploring possible optimizations for scenarios where multiple applications are interested in similar sensor data. We are also interested in using sensor fusion to create even "smarter" sensors that enable the development of context-awake mobile applications.

References

[1] Android 4.4 sdk. http://developer.android.com/about/versions/android-4.4.html.

[2] Android intents and intent filters. http://developer.android.com/guide/components/intents-filters.html.

[3] Moto x. http://www.motorola.com/us/FLEXR1-1/Moto-X/FLEXR1.html.

[4] Nexus 5. http://www.google.ca/nexus/5/.

[5] Qualcomm - 3g/4g connectivity (gobi). https://developer.qualcomm.com/mobile-development/maximize-hardware/3g4g-connectivity-gobi.

[6] BALDAUF, M., DUSTDAR, S., AND ROSENBERG, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing 2*, 4 (2007), 263–277.

[7] BIEGEL, G., AND CAHILL, V. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on* (2004), IEEE, pp. 361–365.

[8] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 49–62.

[9] GELLERSEN, H. W., SCHMIDT, A., AND BEIGL, M. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications 7*, 5 (2002), 341–351.

[10] HAMEED, K. The application of mobile computing and technology to health care services. *Telematics and Informatics 20*, 2 (2003), 99–106.

[11] HONG, J.-Y., SUH, E.-H., AND KIM, S.-J. Context-aware systems: A literature review and classification. *Expert Systems with Applications 36*, 4 (2009), 8509–8522.

[12] HULL, B., BYCHKOVSKY, V., ZHANG, Y., CHEN, K., GORACZKO, M., MIU, A., SHIH, E., BALAKRISHNAN, H., AND MADDEN, S. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (2006), ACM, pp. 125–138.

[13] KANGAS, M., KONTTILA, A., LINDGREN, P., WINBLAD, I., AND JÄMSÄ, T. Comparison of low-complexity fall detection algorithms for body attached accelerometers. *Gait & posture 28*, 2 (2008), 285–291.

[14] LIBBY, R. A simple method for reliable footstep detection in embedded sensor platforms, 2009.

[15] LIN, X., WANG, Z., LIKAMWA, R., AND ZHONG, L. Reflex: Using low-power processors in smartphones without knowing them. *Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2012).

[16] LIU, J., ZHONG, L., WICKRAMASURIYA, J., AND VASUDEVAN, V. uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing 5*, 6 (2009), 657–675.

[17] MAISONNEUVE, N., STEVENS, M., NIESSEN, M. E., HANAPPE, P., AND STEELS, L. Citizen noise pollution monitoring. In *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government* (2009), Digital Government Society of North America, pp. 96–103.

[18] MAISONNEUVE, N., STEVENS, M., NIESSEN, M. E., AND STEELS, L. Noisetube: Measuring and mapping noise pollution with mobile phones. In *Information Technologies in Environmental Engineering*. Springer, 2009, pp. 215–228.

[19] MARR, D., AND HILDRETH, E. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences 207*, 1167 (1980), 187–217.

[20] MATHER, P. M. *Computer Processing of Remotely-Sensed Images: An Introduction*. Wiley, 2005.

[21] MISHRA, N., CHEBROLU, K., RAMAN, B., AND PATHAK, A. Wake-on-wlan. In *Proceedings of the 15th international conference on World Wide Web* (2006), ACM, pp. 761–769.

[22] NATH, S. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 29–42.

[23] PREUVENEERS, D., AND BERBERS, Y. Mobile phones assisting with health self-care: a diabetes case study. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services* (2008), ACM, pp. 177–186.

[24] PRIYANTHA, B., LYMBEROPOULOS, D., AND LIU, J. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE 10*, 2 (2011), 12–15.

[25] RA, M.-R., SHETH, A., MUMMERT, L., PILLAI, P., WETHERALL, D., AND GOVINDAN, R. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (2011), ACM, pp. 43–56.

[26] SCHLÖMER, T., POPPINGA, B., HENZE, N., AND BOLL, S. Gesture recognition with a wii controller. In *Proceedings of the 2nd international conference on Tangible and embedded interaction* (2008), ACM, pp. 11–14.

[27] SHIH, E., BAHL, P., AND SINCLAIR, M. J. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking* (2002), ACM, pp. 160–171.

[28] SORBER, J., BANERJEE, N., CORNER, M. D., AND ROLLINS, S. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSyS)* (Seattle, WA, June 2005).

[29] TSAI, C. C., LEE, G., RAAB, F., NORMAN, G. J., SOHN, T., GRISWOLD, W. G., AND PATRICK, K. Usability and feasibility of pmeb: a mobile phone application for monitoring real time caloric balance. *Mobile networks and applications 12*, 2-3 (2007), 173–184.