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 *Smart Sensors*, a new approach that offloads the periodic 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. Smart sensors 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. Smart sensors 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 smart phones 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 [5, 8], such as targeted advertising, and medical applications [18, 24, 7] 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 [14, 15] monitoring or traffic prediction [9]. 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 [12, 23]. 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 monitoring 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 API should expose access to filter-level functionality. We present *Smart Sensors*, 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, smart sensors provide a better balance between flexibility and ease of deployment. Smart Sensors 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 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 smart sensors could be programmed directly by individual applications, it is easy to imagine user-level activity recognition libraries that encapsulate the functionality of the smart sensor to provide simple wake up conditions for a large number of activities.

To evaluate the benefits of our approach, we have developed a prototype implementation that extends a Nexus 4 phone with a low-power sensor board. Our experiments with various accelerometer-based applications show that:

TODO: summarize results.

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 monitoring, 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 be powered down 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 wakeup 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 [12, 23]. 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

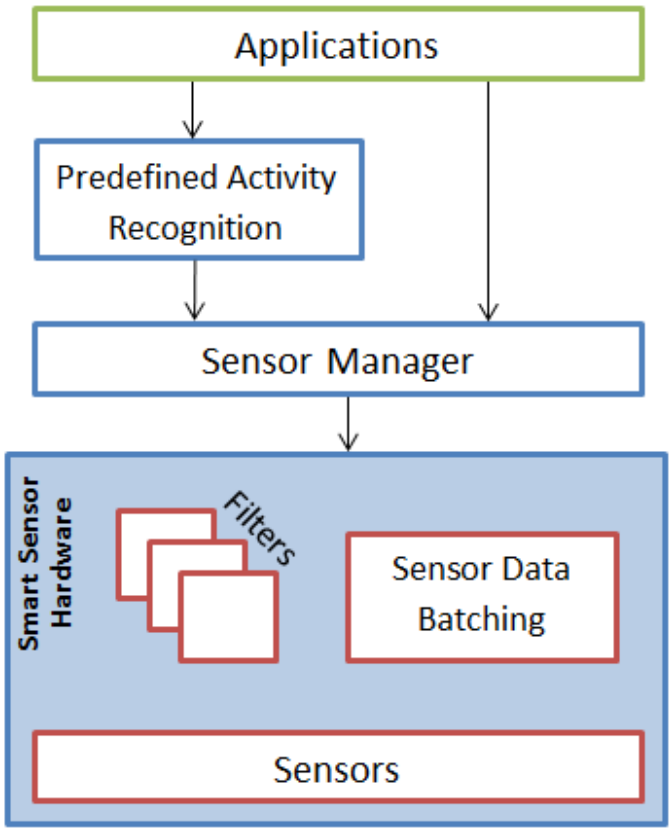

Figure 1: Proposed system architecture using Smart Sensors

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 Smart Sensors

Our aim is to enable low-power, continuous sensing tasks on mobile processors. As described earlier, the most effective approaches for low-power sensing, full computation offloading and activity detection, either complicate programming or have limited generality.

Our approach for low-power sensing takes a middle ground and provides a filter-based sensor API for application programs. Applications create *custom wake up conditions* for events of interest by choosing from a prespecified set of filtering algorithms and tunning 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, as we show later. Since the filters are pre-specified, their implementations can be optimized for each low-power processor. 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 smart sensors 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 smart sensor 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 sensor. 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 smart sensors approach is defining the appropriate set of event filters for each sensor. First, the filters need to be mainly computational tasks. Any code that needs to use resources not available to the low-powered processor, e.g., the graphical user interface, needs to run on the main processor. Second, there is a trade-off in the computational tasks that are carried out by the low-power processor. Pushing additional, more complex computation to the low-power processor has the potential to result in higher accuracy filters, and the main processor will remain asleep for longer periods of time, thus increasing energy savings. However, sensor data processing needs to be performed in real time, and the main processor woken as soon as the event is detected. Pushing too much computation to the low-power processor can overwhelm it. We evaluate this trade-off between filter accuracy and power savings by comparing the behaviour of two low-power processors with different power profiles and computational resources.

We examine the types of computation performed by various sensing applications to determine a common set of filters that are needed for these applications. In our experience, while the application logic varies widely, their sensor event detection algorithms are relatively simple and have commonalities. For example, an accelerometer-based fall detector looks for acceleration that represents free-fall, followed by a spike in acceleration [10]. Similarly, a step detection algorithm looks for a local maxima within a certain range of acceleration [11].

These applications often perform some common initial steps after obtaining sensor readings. 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. 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.

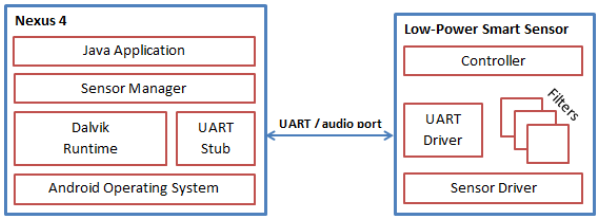

Figure 2: High-level Architecture of the Prototype (Nexus 4 and Smart Sensor board)

4 Prototype

We implemented a prototype of our system using a Google Nexus 4 phone running Android 4.2.2. The low-power smart sensor is implemented using a Texas Instruments micro-controller board attached to an accelerometer sensor. 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. Figure 2 shows a high-level diagram of the prototype's architecture.

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. Nevertheless, the approach can be extended to other low bit-rate sensors such as the microphone, GPS or WiFi. 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.

4.1 Google 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

```
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

define custom wake-up conditions by specifying a predefined data filter and configuring the filter and wake-up parameters. The available types of wake-up conditions, described in detail in Subsection 4.2.1, are a set of predefined implementations of a DataFilterAlgorithm interface. Figures 4 and 3 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 smart sensor board. The UART stub is called when the smart sensor 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 smart sensor, 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 Smart Sensor Board

The low-power smart sensor 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. 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 fast Fourier transformation [11] that is more accurate at removing noise from the sensor data. However, it is computationally expensive. Two parameters can be set for the FFT-LPF: 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.

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: Power Profile for the Google Nexus 4

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 manufactures 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 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 filter in real-time. Stellaris LM4F120H5QR is powered by Cortex-M4 processor. It can batch a higher number of accelerometer readings and 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

Evaluating different sensing methods with human subjects is labour intensive and error prone. Instead, 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). Becuase the robot's actions can be scripted, this setup provides a reliable source of ground truth for comparison.

We developed three applications that detect activities that the robot can perform: walking, seating 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.

Figure 5: AIBO ERS 210 with smart sensor prototype mounted on its back.

Step 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 [11]. 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 seating 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 between $7.5m/s^2$ and $9.5m/s^2$, and the acceleration on the y-axis is between $3.5m/s^2$ and $5.5m/s^2$, 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.

Headbutt detect 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

While our robotic testbed allows for live experiments, we chose instead to use the robot to collect traces and to compare sensing implementations using trace base simulation. We opted for this approach 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 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.

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 axises. 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 set-up 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.

The simulator evaluated each trace for each of the following sensing configurations:

Always Awake Provides a baseline for the following methods in terms of event detection recall and precision.

Duty cycling A software-only solution that checks sensor readings periodically and then puts the phone to sleep. This method can be implemented on any mobile device and does not require special hardware support.

Batching A hardware extension to duty cycling method that matches the Always Awake method in terms of detection recall and precision, but is expected to consume less power.

Activity Assumes hardware support for activity detection, specifically a step detector. Allows the exploration of the benefits of hardware-based activity recognition as a wake-up condition.

Filter Assumes hardware support for the selection of a pre-defined data filter and customization of filter and wake-up parameters. Allows us to explore the benefits of filter customization.

Oracle A hypothetical, ideal wake-up condition. The device only wakes up when there exists an event of interest to be recognized. 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 method using custom filter-based wake-up conditions gives an upper bound on the potential benefits of fully configurable hardware, beyond limited data filter selection.

The simulator replays the collected traces for each of the applications described in Section **??** and each of the above sensing strategies, and determines the the amount of time the smartphone is awake and asleep and the total number of wake-ups. In addition, for each application the simulator determines its recall [1] and precision [2]. Finally, using an energy model derived from measurements of our smartsensor prototype, the simulator estimated the average power consumption of each configuration. For *Always Awake* and *Duty Cycling*, the power model accounts only for the energy consumption of the Nexus 4. For *Batching* and *Activity* the model also includes the cost of the simple TI MSP430 micro-controller. Finally, *Filter* and *Oracle* include the cost either the TI MSP430 or TI Stellaris LM4F120H5QR, depending on the condition being evaluated.

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

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

6 Results

This section answers the following questions:

1. Are there applications/usage scenarios that are supported efficiently with duty cycling. What are the power saving achieved by this approach?
2. What are the power savings achievable with the batching wake up approach?
3. Under what usage scenario does the customizable wakeup condition approach improve performance (over the other approaches), and by how much?
4. What are the benefits of having customizable wakeup conditions? Do different applications require different wake up conditions or is a static wake-up condition sufficient for all applications?
5. How much additional benefit can be obtained from fully programmable wake-up conditions over customizable wake-up conditions based on data filtering and thresholding?
6. Is the best sensor data filter choice dependant on the type of event of interest?
7. How sensitive are detection recall and energy savings to the configuration thresholds used in the wake-up condition?
8. Are these results similar for human experiments?

Ashvin: You need to mention earlier in the paper that you will be doing some evaluation for human traces? That was not mentioned earlier and so this comes as a surprise

Ashvin's: Briefly restate each question at the beginning of each subsection

6.1 Energy Savings

This section answers the first three questions. Table 2 summarizes the power and event of interest recall data for the each of the wake-up approaches tested in our experiments.

Duty cycling has significant drawbacks compared to other wakeup approaches. Using this approach with a sleep interval of 2 seconds actually resulted in an increase in power consumption compared to the Always Awake approach and a decrease in detection recall. This is a result of frequent transitioning between the awake and asleep states. Using sleep intervals greater than 5 seconds, we were able to obtain significant power savings. However, such long wake-up intervals caused more than a 50% decrease in event of interest recall for infrequent events (Headbutts and Transitions) and more than 15% decrease for step detection. We also note that in cases where the event of interest occurs rarely (group 1), the power savings of duty cycling are suboptimal because of numerous wakeups that result in no events of interest being detected. We conclude that duty cycling is not appropriate for any type of application that requires high detection recall.

Batching achieves the same event of interest recall as the Always Awake scenario, and is able to reduce power consumption for sleep intervals greater than 5 seconds. Shorter wake- up intervals can potentially cause more energy consumption than the Always Awake case because of the additional energy cost of transitioning between the sleep and awake states. Note that increasing the wake-up interval has diminishing power saving returns. A lower bound to the power consumption of the entire system is the sum of the power consumed by the main device while sleeping and the peripheral processor while being awake. Compared to the other approaches, batching has the best power savings for use cases where the event of interest occurs very frequently (i.e. walking in group 3). However, if the event of interest occurs very infrequently, batching suffers from the same problem as duty cycling. Its power consumption becomes suboptimal because of a large number of wake-ups that do not result in an event of interest being detected. Table **??** shows that Batching has the same power consumption regardless of activity-level (group 1 versus group 3). In this approach, events of interest are only detected after the main processor wakes up. As a result, the delay between the time when the event is occurring and the time when it is detected can be as long as the batching interval. Therefore, this approach may not be appropriate for applications that are expected to react immediately after an event of interest occurs. For example, the user of a gesture recognition application [13, 21] would not be satisfied if the application detects the performed gesture after a delay of more than a couple of seconds. We note that the maximum batching interval is limited by the amount of local storage available to the hardware sensor. While theoretically this approach can achieve very high energy savings (e.g. waking up every few hours), in practice realistic sleeping times are in the order of a few seconds.

When comparing the power consumption for the Duty Cycling and Batching approaches for a specific sleep interval, we need to note several things. First off, the sleep interval also includes the time spent transitioning between the awake and asleep states and vice-versa. Secondly, after a wake-up Duty Cycling keeps the phone awake for at least 4 seconds in order to collect sensor data. Batching on the other hand, only keeps the device awake for just enough time to transfer the entire batch of sensor data from the hardware sensor and to allow the application(s) to process the sensor readings. We estimated this time to be about 10% of the sleep interval.

Table 2 includes the data for the customizable filter-

	Approach	Sleep Interval (seconds)	Power (mW)			Recall		
			Walking	Headbutt	Transitions	Walking	Headbutt	Transitions
Group 1	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 Detection		48.3			98%	36%	87%
	Filter-based Wake-up Condition		48.3	60.3	18.6		100%	100%
	Oracle Wake-up Condition		41.5	59.6	16.4			
Group 3	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 Detection		321			97%	95%	86%
	Filter-based Wake-up Condition		321	65.7	51.7		100%	100%
	Oracle Wake-up Condition		266	62.9	34.9			

Table 2: Summary of achieved recall and power consumption for each sensing approach

based wake-up condition approach. Table 3 lists the data filter and threshold parameters that resulted in the best performance; lowest power consumption while matching the detection recall of the Always Awake approach. This approach achieved the lowest power consumption in every scenario other than step detection in group 3. All these scenarios are similar in that the event of interest occurs infrequently (less than 15% of the time). Walking represents about 63% of the courses in group 3. For this scenario, only the batching approach achieved better power savings.

6.2 Predefined Action vs Predefined Filter

This section answers question 4. For this purpose, we assume that step detection is the one action that our hardware supports and we compare that with the benefits of custom filters for each of the other two applications.

We start off by selecting the best wake-up condition for step detection (EMA-LPF data filter with $alpha =$ 50% and a x-axis minimum threshold of $2\,m/s^2$, as shown in Table 3). We use this wake-up condition and attempt to detect headbutts and stance transitions. The results in terms of detection recall and power consumption are included in Table 2. We compare these values with the results for the best filter-based wake-up condition.

We note that for both the Headbutt and Transition actions, using a custom filter-based wake-up condition increased detection recall to 100% and in most cases reduced power consumption. For group 3, the wake-up condition used for step detection resulted in 95% of headbutts being detected. However, this required almost 400% additional power compared to the best filter-based wake-up condition for headbutts. For transitions, using a custom filter reduced power consumption by at least a factor of 2.5. In terms of power consumption the best results are visible in group 3, the group with the highest amount of walking. In group 3, the energy consumption was reduced by approximately 84%.

Action	Data Filter	Threshold(s)
Walking	EMA-LPF (alpha = 50%)	x-axis $\geq 2m/s^2$
Headbutts	FFT-LPF (relative energy threshold = 10%)	y-axis $\leq -3.5m/s^2$
Transitions	EMA-LPF (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 3: Filter-based Wake-up Conditions parameters that achieved the lowest power consumption while matching detection recall of the Always Awake approach

We can conclude that using a fixed wake-up condition (or predefined hardware supported action detection) may work well for a small set of application, but is inefficient for most other applications. As such, developers should be able to configure the wake-up conditions based on the needs of their applications.

6.3 Customizable filter-based wake-up condition vs Oracle

This section answers question 5. We wanted to examine how much additional benefit can be obtained from fully programmable wake-up conditions over the customizable wake-up conditions based on data filtering and thresholding. A fully-programmable wake-up condition has the potential to be "perfect". That is, it will only wake up the device if and only if an event of interest will be detected by the application (i.e. 100% wake-up precision). Such a wake-up condition will achieve the lowest possible power consumption while matching the detection recall of the Always Awake approach. We will refer to this wake-up condition as the Oracle. Such an wake-up condition could be implemented by moving most of the functionality of the application to the low-power processor. However, this approach has significant limitations, as described in Section 2.4.

We used the list of events detected by the Always Awake simulations to determine the awake time, asleep time and number of wake-ups for the Oracle wake-up condition. These values were then used to estimate power consumption. Table 4 compares the power compunction of the Oracle with the power consumption for the best custom filter-based wake-up condition presented earlier. The difference between these values represents how much additional power savings can potentially be achieve by allowing developers full-programmability of the wake-up conditions. Table 4 also presents this difference as a percentage of the power consumption of the Always Awake approach. For most usage scenarios, having an Oracle wake-up condition results in less that 5.5% additional power savings. For high-frequency events (walking in groups 2 and 3), the Oracle has significantly more power savings, 10.9% and 17.0%, respectively.

6.4 Sensor Data Filter Choice

This section answers question 6. Table 5 lists the best achieved power consumption by the Custom filter-based Wake-Up Condition approach using each of the sensor data filters, while matching the detection recall of the Always Awake approach. We observe that the best and worst sensor data filter changes depending on the event of interest. The Null filter had the second best power consumption for all of the applications. The EMA-LPF has the lowest power consumption for the Walking and Transitions applications, while the FFT-LPF had the lowest power consumption for the Headbutts application.

We observe that different data filters are optimal for different applications. Even in cases where the same filter type is optimal, the configuration parameters for the filter and the wake-up thresholds may be different. Additionally, applications might only be interested in the acceleration values on specific axes, preventing the same filter from being shared with other applications.

6.5 Wake-up Threshold Sensitivity

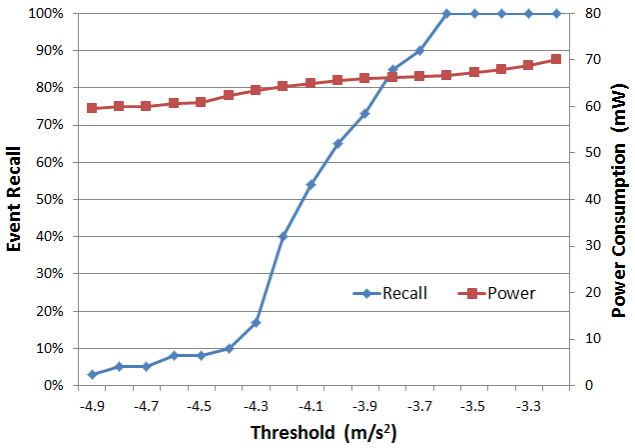

Figure 6: Recall and Power Consumption at various wake-up thresholds for the Headbutt application

This section answers question 7. Figure 6 shows the detection recall and power consumption for the Headbutt application using the best wake-up condition. Simi-

	Action	Recall	Power Usage Oracle Wake-up Condition	Power Usage Custom Wake-up Condition	Additional savings by Oracle	Savings as % of Always Awake
Group 1	Walking	98%	41.5	48.3	6.8	2.1%
	Headbutts	100%	59.6	60.3	0.7	0.2%
	Transitions	100%	16.4	18.6	2.2	0.7%
Group 2	Walking	99%	153	188	35	10.9%
	Headbutts	100%	62.6	65.1	2.5	0.8%
	Transitions	100%	29.5	43.3	13.8	4.3%
Group 3	Walking	97%	266	321	55	17.0%
	Headbutts	100%	62.9	65.7	2.8	0.9%
	Transitions	100%	34.9	51.7	16.8	5.2%

Table 4: Comparison of power savings using a perfect Wake-up Condition (Oracle) versus the best custom filter-based wake-up condition

Sensor Data Filter	Power Consumption (mW)		
	Walking	Headbutts	Transitions
Null	191	186	67.8
EMA-LPF	188	228	43.3
FFT-LPF	217	65.1	88.7

Table 5: Summary of best achieved power consumption for each of the sensor data filters for Group 2

lar graphs were generated for the other applications, but were omitted due to lack of space. We note that the choice of threshold is important, as choosing a threshold that is too strict will cause a significant drop-off in the achieved recall. However, choosing a threshold that is too lenient will result in additional power consumption without any extra benefit to recall because of unnecessary wake-ups.

6.6 Macrobenchmarks

This section answers question 8. We collected three accelerometer traces of three different individuals performing routine daily activities. The first trace is 2 hours and 10 minutes long captured during an individual's morning commute. The second trace is 1 hour and 2 minutes long and captured by a person working in a retail store. The third trace is 3 hours and 17 minutes long, captured by a person working in an office environment. Roughly 20% to 45% of each trace is spent walking. This usage scenario is similar with walking in groups 2 and 3 for the microbenchmarks.

Table 6 shows a summary of the results obtained by running the simulator using these traces. Since ground truth information is not available, detection recall is determined by comparing the steps detected by each wake-up approach with those detected by the Always Awake approach.

The Always Awake approach is used as a baseline. The estimated power consumption for this approach is 323 mW.

The results for Duty Cycling are almost identical to the results of the microbenchmarks. Using a sleep interval of 2 seconds results in a detection recall above 95%, but it uses more power than the Always Awake approach because of frequent device wake-ups. Increasing the sleep interval results in some power savings, but at a significant cost to detection recall.

The results for Batching and Custom Filter-Based Wake-up Conditions are similar to the microbenchmarks results for step detection in group 3, where the event of interest was very frequent. Both Batching and the best Wake-up Condition match the performance of the Always Awake approach (detecting the same steps) and both have significant power savings. The lowest power consumption was achieved by batching with a sleep interval of 30 seconds. The best custom filter-based wake-up conditions also achieved more than 55% power savings in each of the traces. Batching has the best performance because in only requires about 10% awake time, compared to wake-up conditions for which the awake time is proportional to the frequency of events on interest. However, batching introduces a delay between the time when the event occurs and the time when it is detected by the application. While this drawback is tolerable for the Pedometer application used in our experiments, it may be unacceptable for other types of applications.

We conclude that these results are similar to their microbenckmark counterparts.

Approach	Sleep Interval (seconds)	Power (mW)			Average Recall
		Trace 1 35% - 45% walking	Trace 2 20% - 30% walking	Trace 3 20% - 30% 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%
Wake-up Condition		136	77.9	72.6	100%

Table 6: Summary of achieved recall and power consumption for each wake-up approach for macrobenchmarks

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. TODO: Telephony. Wake on Wireless [22] extended this idea by augmenting a PDA with a low-power radio that would send a wakeup message when an incoming call is received. Similarly, Wake on WLAN [16] allows remote wakeup of wireless networking equipment.

Turducken [23] 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 [19] applies Turducken's multi-tiered architecture to sensing on mobile devices. Reflex [12] complements the idea proposed by Turducken by providing a shared memory abstraction to be used by the different processors.

Smart Sensors 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 Smart Sensors 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.

Recently, smartphone manufacturers have started incorporated 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 of a small number of predefined activities that can be used as wake up conditions [3].

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 [17] 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 Smart Sensors architecture (once we implement support for multiple sensors) and achieve additional power savings.

While our focus was on power-efficient acquisition of sensor data, next generation mobile perception applications face a related problems that are the active focus of other research. MAUI [6] enables fine- grained energy-aware offload of mobile application code to remote servers. Similarly, Odessa [20] uses code-offloading to address the issue of processing excessive amounts of sensor data on resource constrained mobile devices.

8 Conclusion

TODO

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] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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).

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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).

[24] 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.