SmartSensors: Energy-efficient sensing on mobile devices

Silviu Jingoi, Wilson To[†], Italo De Moraes Garcia, Kevin Lee, Eyal de Lara and Ashvin Goel[†]
Department of Computer Science
[†]Department of Electrical and Computer Engineeering
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, such as targeted advertising, and medical applications 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 pollution monitoring or traffic prediction. 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 [7, 8]. 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 [7, 8]. 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. 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.

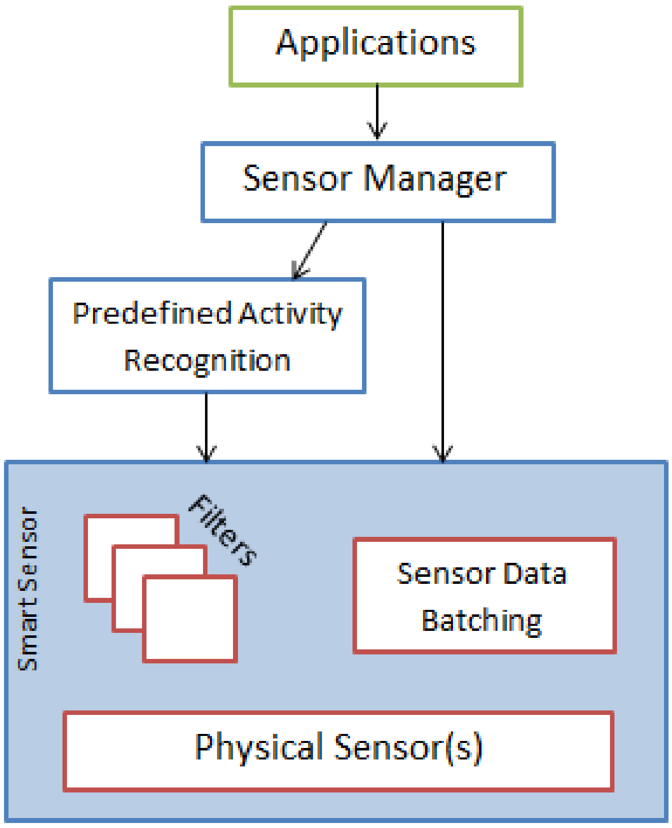

Figure 1: Proposed system architecture using Smart Sensors

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 pre-specified 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 [5]. Similarly, a step detection algorithm looks for a local maxima within a certain range of acceleration [6].

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.

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.

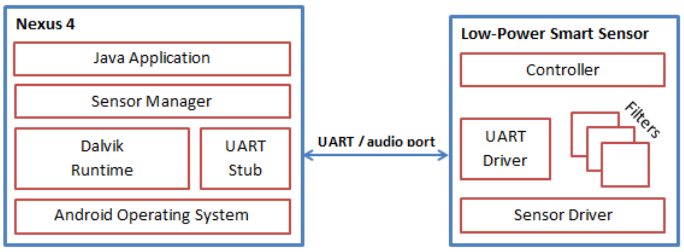

Figure 2: High-level Architecture of the Prototype (Nexus 4 and Smart Sensor board)

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. Normally, in order to receive sensor data, an application needs to register a listener (i.e. an implementation of the SensorEventListener interface) for a specific sensor with Android's SensorManager. Figure 3 shows a code example of a typical application registering a listener with the Android SensorManager for accelerometer readings. We extended the Sensor Manager to allow users to define custom wake-up conditions by specifying a pre-defined data filter and configuring the filter and wake-up parameters. The available types of wakeup conditions, described in detail in Subsection 4.2.1, are a set of predefined implementations of a DataFilterAlgorithm interface. Figure 4 shows an example of the modified code to include an exponential moving average low pass filter with an alpha value of 75% and a wakeup condition that is satisfied when the filtered x-axis acceleration exceeds 12 m/s^2.

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 wakeup condition.

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

Figure 3: Typical usage of Android's SensorManager

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

Figure 4: Usage of the SensorManager with a wakeup 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 and possibly modifications to the kernel.

We power profiled the Google Nexus 4. The results are summarized in Table 1. During all the measurements, the device's screen was turned off. Additionally, we noticed that changes in signal strength for GSM, WiFi and GPS resulted in fluctuations in the power consumption of the device. To prevent these factors from affecting our power profile, we decided to run all the power measurements with GSM, WiFi and GPS 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 wakeup transition, the average power consumption goes up to 384 mW, while during an awake-to-asleep transition the average power consumption is 341 mW.

TODO for Wilson: we probably need a section that describes how the UART stub opens a serial port and then uses a TTY to copy the sensor readings and start the shell script that notifies the sensor manager. Talk about the bandwidth that we get and how long it takes us to copy sensor readings.

4.2 Smart Sensor Board

On the low-power smart sensor board, the prototype 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 wakeup condition is satisfied.

4.2.1 Sensor Data Filters

Accelerometer sensor data is particularly noisy. As a result many applications using this kind of data implement low-pass filters to remove noise and "smooth out" the accelerometer readings. On our Smart Sensor board we implemented three types of sensor data filters of varying complexities.

The **Null Filter** algorithm is the simplest and requires no additional computation. This algorithm performs no filtering (i.e. the output sensor data is the same as the input sensor data). We use this "filter" to pass raw accelerometer readings to algorithms down the pipeline.

The **Exponential Moving Average Low-Pass Filter** (EMA-LPF) removes some of the noise from the sensor data and is computationally cheap, requiring only a few algebraic operations for every sensor reading. This filter takes an alpha parameter between 0 and 1, that controls the "smoothness" of the filtered data. This filter can be described using the following equation:

$$\textit{output reading} =$$
$$= (1 - \alpha) \times \textit{last output reading} + \alpha \times \textit{input reading}$$

The **Fast Fourier Transformation based low-pass filter** [6] (FFT-LPF) 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.

4.2.2 Wake-up conditions

In our prototype, a wake-up condition comprises of

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

a sensor data filter as described in the previous section and a set of wake-up parameters. This set contains the type of threshold to be used and the threshold value(s). We implemented three similar types of thresholds:

- A **Minimum Threshold** that is satisfied when the acceleration exceeds the threshold value
- A **Maximum Threshold** that is satisfied when the acceleration goes below the threshold value
- A **Threshold Range** that is satisfied when the acceleration is between two threshold values

4.2.3 Hardware Options

TODO for Wilson: We need to talk about the actual sensor used by the board.

We evaluated two options for our low-powered sensor platform.

Our first option is a Texas Instruments MSP430 microcontroller. It 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, the MSP430 was unable to run the Fast Fourier Transformations necessary for low-pass filtering sensor data in real-time.

Our second option is a Texas Instruments Stellaris LM4F120H5QR micro-controller powered by a Cortex-M4 processor. It can batch a higher number of accelerometer readings and run all our wakeup conditions in real-time, including the one using FFT-based low-pass filtering. 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.

5. EXPERIMENTAL SETUP

Our evaluation compares six different continuous sensing methods:

1. **Always Awake**. Provides a baseline for the following methods in terms of event detection recall and precision.
2. **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.
3. **Sensor data 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.
4. **Activity detection**. 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.
5. **Custom filter-based wake-up condition**. Assumes hardware support for the selection of a predefined data filter and customization of filter and wake-up parameters. Allows us to explore the benefits of filter customization.
6. **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.

We implement applications using these different sensing methods and use trace-based simulation for comparison. The rest of this section describes our trace collection methodology, the sensing applications that we have implemented, and the operation of our trace-based simulator.

5.1 Trace Collection

We use traces in our evaluation because they help provide the same input to the different sensing methods. To compare these methods, we need ground truth about the occurrence and the timing of desired events, such as steps taken. Annotating accelerometer traces collected from human subjects with the ground truth is time-consuming and error prone.

Instead, we used an AIBO ERA 210 robot dog (see Figure 5) to collect the ground truth about events and accelerometer traces for our experiments. The robot performed a sequence of actions in a course, and we logged these actions and the timestamps for the beginning and the end of each action. This log represents the ground truth for our experiments. We also logged the robot's acceleration by attaching a Google Nexus 4 to the back of the robot. The smartphone was used to run an application that kept the device continuously awake and recorded all the accelerometer readings. The accelerometer trace collected for each course is an ordered set of accelerometer readings. Each reading contains the x, y and z components of the acceleration vector and a timestamp indicating when the reading was produced.

To get statistically significant results, the robot was programmed to run multiple different courses. In a course, the robot performed five different types of actions: standing idle, walking, sit-to-stand transitions, stand-to-sit transitions, and headbutts. We divided the courses into three different groups to cover an increasing amount of activity levels. Courses 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. Figure 6 shows the average amount of time that each action is performed in each group, as a percentage of the average time taken to complete the course. This set-up allows us to experiment with detecting actions that are common, somewhat frequent, and rare. In total, the robot ran 18 different courses, partitioned as follows: 9 for the group 1, 6 for group 2 and 3 for group 3. We chose to run more course in 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 course, based on the expected probabilities of each action occurring. Running each course generated a separate ground truth log of actions and an accelerometer trace.

5.2 Applications

The objective of this work is to match the recall and precision of the Always Awake approach while using less power. Our objective was not to build more accurate sensor-driven applications, but to lower the energy cost of running these kinds of algorithms. We present detection precision and recall metrics to demonstrate that the applications we used are doing a reasonable job, and are therefore legitimate choices to use in our experiments. Achieving perfect event of interest detection is not our objective.

Ashvin: as stated, this sentences sounds like we carefully chose applications that would work well for us (thus introducing bias), and they do "okay" with our approach. consider dropping or rewording this entire sentence. e.g., it is not clear what point the sentence is making. The sentence (achieving perfect ..) could also be dropped

For our microbenchmarks, we implemented three applications:

The **Step** application 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 [6]. 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.

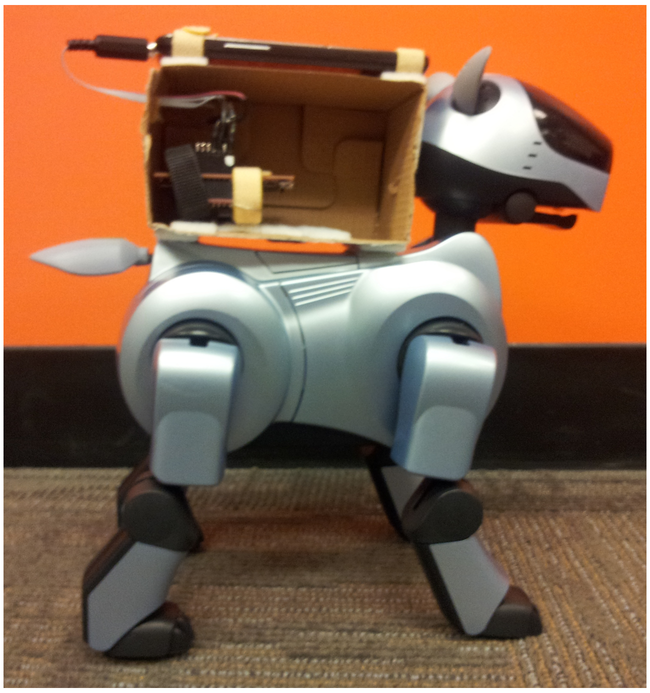
Figure 5: AIBO ERS 210 robot used to collect the accelerometer traces

The **Headbutt** application works very similarly to the Steps application. The major differences are that it uses the y-axis acceleration and that it searches for local minima between $-3.75\,m/s^2$ and $-6.75\,m/s^2$.

The **Sit-Stand** 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, as shown in Figure 5. 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.

Figure 7 shows the average recall for all applications when executing on a phone that is kept always on. The Headbutt and Stand-Sit applications detected all the

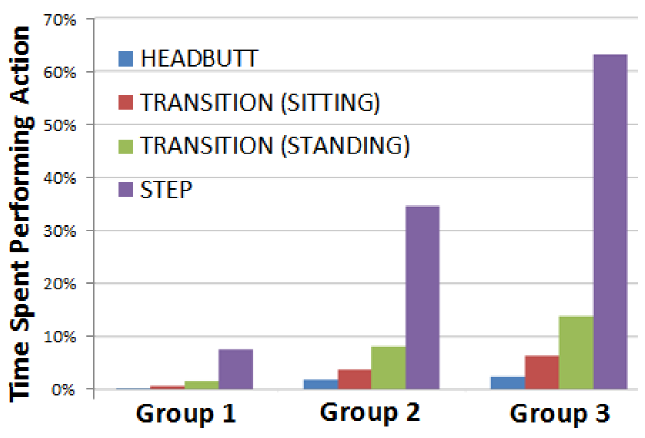

Figure 6: Percent time spent performing actions of specific types

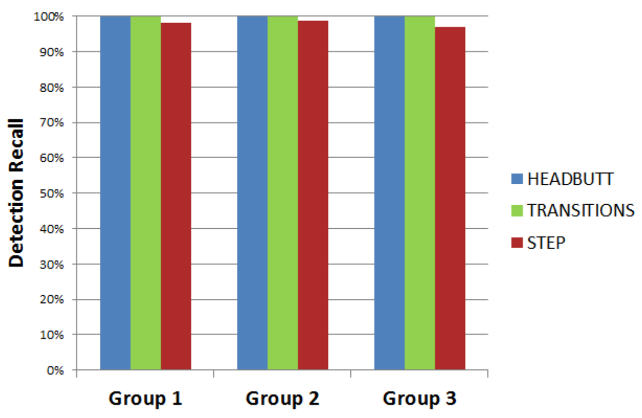

Figure 7: Always Awake: Recall by Group

events of interest, achieving a recall of 100%. The Steps application achieved a recall above 97% for all three groups. Figure ?? shows the detection precision of each of the events of interest averaged over the traces in each group. For all cases, the applications achieved a detection precision above 82%.

5.3 Trace-based Simulator

The objective of our simulations is to compare the different sensing methods in terms of detection recall, wakeup precision and power consumption for various usage scenarios. To do this, the simulator is used to process the collected traces using each of the sensing methods listed at the beginning of this section. Also, the simulator allows us to explore the parameter configuration space for each of the more complex sensing methods (duty cycling, batching, custom wake-up conditions).

Each simulation run takes several inputs:

- An accelerometer trace. These files were collected on the smart phone attached to the robot during each of the courses.
- A ground truth log file. These files were generated by the robot while performing each of the courses.
- The sensing method to be used.
- A set of parameters for the sensing method used.

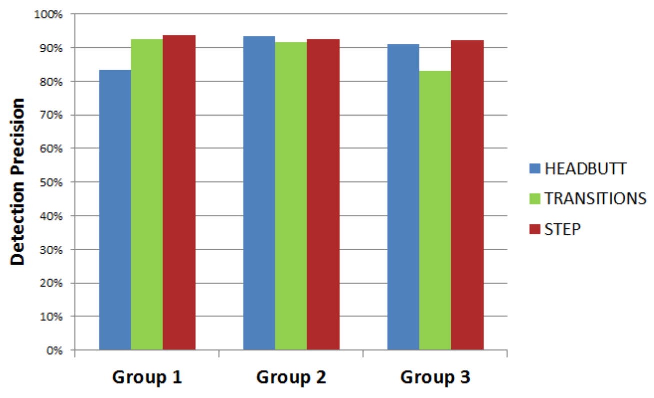

Figure 8: Always Awake: Detection Precision by Group

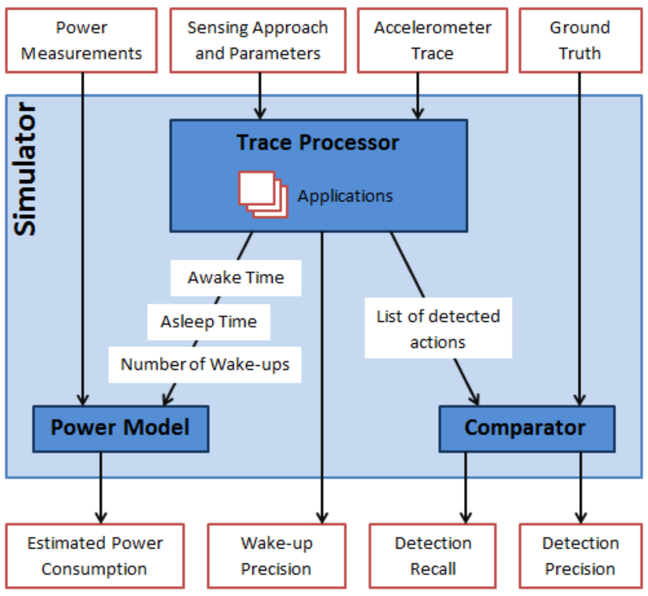

Figure 9: Overview of the Simulator and its three main modules

Figure 9 shows an overview of the simulator. The simulator is made up of three main modules: a *Trace Processor*, a *Comparator* and a *Power Consumption Model*.

The *Trace Processor* simulates both the low-power and the main processor. It takes the accelerometer trace and handles the readings one at a time, similar to the way a mobile device would receive accelerometer readings in real-time, as they are produced by the sensor. Based on the sensing method used and its parameters, the *Trace Processor* determines when the device should wake-up and what accelerometer readings should be sent to the running application(s). This enables the *Trace Processor* to output the amount of time the device is awake and asleep, the total number of wake-ups and the wake-up precision (defined below). Also, the *Trace Processor* runs the detection applications and produces a list of actions detected by the applications and the times when each of the actions occurred.

The *Comparator* takes as inputs the file containing ground truth data about the actions performed by the robot during the course and the list of detected actions that is produced by the *Trace Processor*. It compares the two lists based on action timestamps and determines detection recall and precision.

Finally, the *Power Consumption Model* module uses metrics from the *Trace Processor* to estimate the average power consumption of the system. The input metrics are awake and asleep times of the device, and the total number of wake-ups. The *Power Consumption Model* estimates power consumption based on the power consumption measurements presented in Subsections 4.1 and 4.2.

Overall, the main metrics outputted by the simulator are:

Move Table 3 after Table 2. What about Group 2? Why not present Precision as well. It is hard to gauge absolute power numbers. Why not present the ratio power over power in always awake

- Detection recall of event of interest

$$Recall = \frac{Correctly\ detected\ actions}{Number\ of\ actions\ in\ ground\ truth}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

- Detection precision of event of interest

$$Precision = \frac{Correctly\ detected\ actions}{Number\ of\ detected\ actions}$$

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

- Estimated average power consumption over the duration of the trace

- Wake-up precision

$$Wakeup\ Precision =$$
$$= \frac{Number\ of\ wakeups\ with\ detected\ actions}{Number\ of\ wakeups}$$

5.4 Discussion

Since we are mainly interested in actions that would normally be performed by humans, we configured the robot to perform actions with similar acceleration signatures. 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.

Rather than using a trace-based simulator, it is possible to run live experiments with a robot because it can perform a deterministic sequence of actions multiple times. However, we chose to use the simulator for several reasons. First, each course takes more than an hour. Our goal was to obtain results for the various sensing approaches, and perform an exhaustive exploration of the parameter configuration space. The combination of multiple courses, wake-up and parameter configurations would have required over a year of continuous live experimentation. Moreover, taking fine grain power consumption measurements while the robot is in motion is not trivial. Nonetheless, we plan to validate our simulation results using live experimentation in the future.

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. How dependent are the best filter/threshold combinations on the usage scenario?
7. Is the best sensor data filter choice dependant on the type of event of interest?
8. How sensitive are detection recall and energy savings to the configuration thresholds used in the wake-up condition?
9. 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

	Action	Power (mW)	Recall
Group 1	Walking	48.3	98%
	Headbutts	60.3	100%
	Transitions	18.6	100%
Group 3	Walking	321	97%
	Headbutts	65.7	100%
	Transitions	51.7	100%

Table 3: Summary of achieved recall and power consumption for the Wake-up Conditions approach

6.1 Energy Savings

This section answers the first three questions. Table 2 summarizes the power and event of interest recall data for the Always Awake scenario and for various wake-up intervals for the Duty Cycling and Batching approaches. Similarly, Table 3 summarizes the results for the approach that uses wake-up conditions. The information in this figure represents the best wake-up conditions in terms of power consumption that achieved the same recall as the Always Awake scenario.

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 2 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 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.

Using the customizable filter-based wake-up condition approach we were able to match the detection recall of the Always Awake approach and significantly reduce power consumption in most scenarios. In fact, 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, as seen in Figure 6). Walking represents about 63% of the courses in group 3. For this scenario, only the batching approached achieved better energy savings. Table 4 lists the data filter and threshold parameters that resulted in the best performance. Wake-up conditions with these parameters had the lowest power consumption while matching the detection recall of the Always Awake approach.

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

	Approach	Sleep Interval (seconds)	Power (mW)	Recall		
				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%
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%

Table 2: Summary of achieved recall and power consumption for the Always Awake, Duty Cycling and Batching approaches

Action	Data Filter	Threshold
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%)	Range Thresholds $2.25m/s^2 \leq$ y-axis $\leq 3.25m/s^2$ $8.2m/s^2 \leq$ z-axis $\leq 9.2m/s^2$

Table 4: Filter-based Wake-up Conditions parameters that achieved the lowest power consumption while matching detection recall of the Always Awake approach

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 4). We use this wake-up condition and attempt to detect headbutts and stance transitions. The top part of Table 5 shows the results in terms or detection recall and power consumption. In the bottom half of the table we present the results for the same metrics when we used the best wake-up condition for those specific actions (see Table 4).

We note that in every single case, using a custom filter increased detection recall and in most cases reduced power consumption. For the headbutt action, recall more than doubled for groups 1 and 2. For group 3, the wake-up condition used for step detection resulted in all headbutts being detected. However, this required almost 400% additional power usage compared to the best custom wake-up condition for headbutts. For transitions, using a custom filter increased recall to 100% for every group and 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%.

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.

Wake-up condition used		Headbutts			Transitions		
		Group 1	Group 2	Group 3	Group 1	Group 2	Group 3
Best for step detection	Recall	35.7%	13.0%	100%	87.5%	72.6%	86.0%
	Power	48.3	177	321	48.3	177	321
Best custom	Recall	100%	100%	100%	100%	100%	100%
	Power	48.6	65.1	65.7	18.6	43.3	51.7

Table 5: Detection recall and power consumption for the Headbutts and Transitions actions using fixed or custom wake-up conditions

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 6 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 6 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 Usage Dependence

This section answers question 6. Figure 10 shows the step recall based on the wake-up condition threshold value for each of the groups. We note that for different usage scenarios (i.e. each of the groups), the recall does not change by more than 10% for any given threshold value. Similar results were found for all the events of interest.

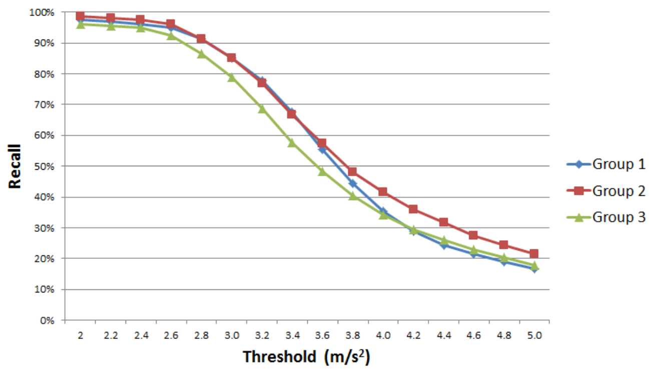

Figure 10:

We conclude that the best filter and threshold combinations are not dependent on the usage scenario.

6.5 Sensor Data Filter Choice

This section answers question 7. Table 7 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.

6.6 Wake-up Threshold Sensitivity

This section answers question 8. Figure 11 shows the detection recall and power consumption for the Headbutt application using the best wake-up condition. Similar graphs were generated for the other applications, but were omitted due to lack of space. We note that the

	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 6: 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 7: Summary of best achieved power consumption for each of the sensor data filters for Group 2

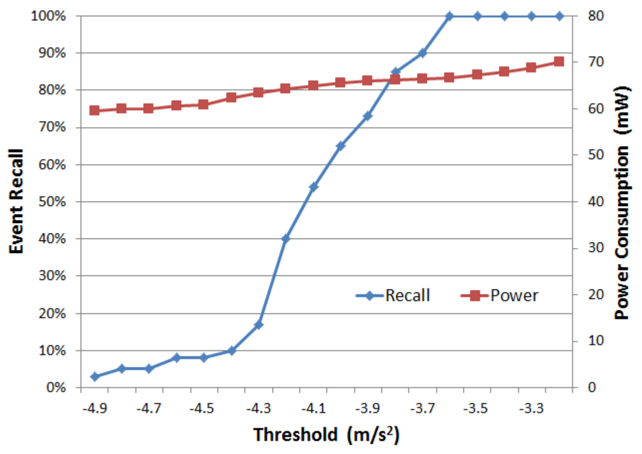

Figure 11:

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.

6.7 Macrobenchmarks

This section answers question 9. 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 8 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-

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

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 unaccptable for other types of applications.

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

7. RELATED WORK

TODO

8. CONCLUSION

TODO

9. 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] M. Kangas, A. Konttila, P. Lindgren, I. Winblad, and T. Jämsä. Comparison of low-complexity fall detection algorithms for body attached accelerometers. *Gait & posture*, 28(2):285–291, 2008.

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

[7] X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. 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.

[8] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. 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.