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 [4, 5]. 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, 2, 3].

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 sensor let developers 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 **??** 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 On

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 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. First, events of interest will not be detected if they occur while the device is sleeping because the application does not receive any sensor data that were produced at those times. Second, events of interest may not occur during the short time that the device is awake. 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 behavior. 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 detect accuracy similar to the power-on 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.

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 [4, 5]. This approach allows developers to offload their own sensor data analysis algorithms to the peripheral processor, enabling arbitrary 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 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 [2] 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.

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 have filters in common.

Figure 1 shows the logical 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 avialable pre-defined filters. The figure also shows the that 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 impemented by defining separate indepedent 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 computation to the low-power processor will generally result in higher accuracy filters, and the main processor will remain asleep for longer periods of time, thus increasing energy savings. However, this computation needs to be performed on the low-power processor in real time, and the main processor woken as soon as the event is detected. We evaluate this trade-off between filter accuracy and power savings by comparing the behavior of two different low-power processors.

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 [TODO:cite]. Similarly, a step detection algorithm looks for a local maxima within a certain range of acceleration [TODO:cite].

These applications often perform some common initial steps. 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 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. Figure 2 shows a high-level diagram of the prototypes 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 sensors such as the microphone, GPS or WiFi. However, extending the prototype to work with higher bandwidth 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 include functions that take an additional parameter that specifies which wake-up condition to be used. The available types of wakeup conditions, described in detail in Subsection 4.2.1, are a set predefined implementations of a WakeUpCondition interface. Figure 4 shows an example of the modified code to include a wakeup condition that is satisfied when the raw x-axis acceleration exceeds $12m/s^2$.

We also created a UART stub to facilitate the 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. In turn, the UART stub notifies the SensorManager using an Android Intent(TODO: reference to Android Intent documentation?).

For our prototype we did not have to modify the operating system on the phone 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 very likely make use of the higher bandwidth bus such as the I^2C and require a custom driver and possibly modification to the operating system.

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 consumption 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 rises to 341 mW.

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. The UART driver is in charge of waking up the phone, running a shell program that issues a Android Intent to start the UART stub, and passing the sensor readings to the main device.

4.2.1 Sensor Data Filters

We implemented three types of sensor data filters (or wakeup conditions) of varying complexities. All wakeup conditions are based on thresholding over some accelerometer readings. The wakeup conditions differ in what readings are checked against the threshold value.

The *raw acceleration wakeup condition* is the sim-

```
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);
WakeUpCondition mWakeUpCondition = new MinThresholdWakeUpCondition(Sensor.AXIS_X, 12.0);
mSensorManager.registerListener(new MySensorEventListener(), mAccelerometer, mWakeUpCondition);
```

Figure 4: Usage of the SensorManager with a wakeup condition

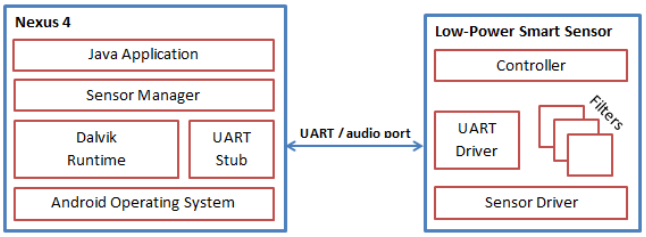

Figure 2: High-level architecture of our prototype implementation

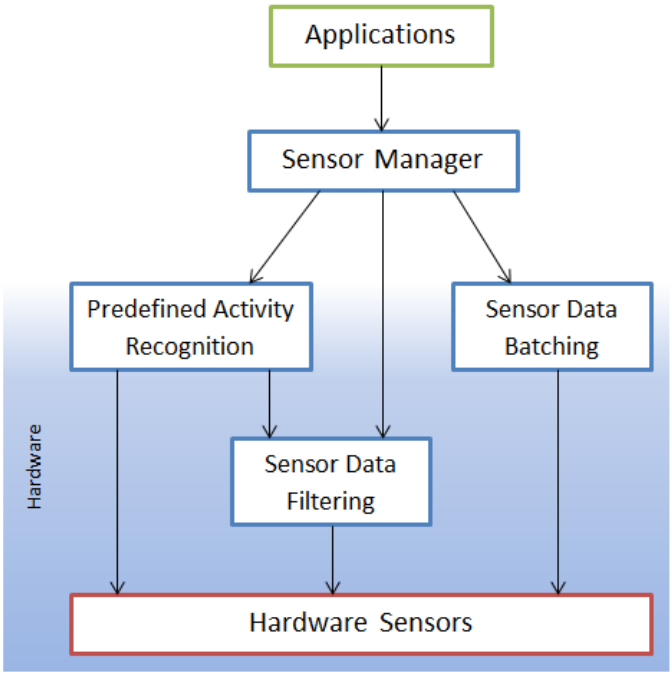

Figure 1: Smart sensor architecture

plest and requires the least amount of computation. The wakeup condition is satisfied if the raw acceleration exceeds the threshold. The threshold value is the only parameter for this wakeup condition. In our preliminary tests, we noticed that accelerometer data is particularly noisy. Our expectation was that a simple threshold filter over raw acceleration readings would be satisfied more frequently than necessary because of the noise present in the sensor data, potentially causing unnecessary device wake ups.

The *EMA LPF wakeup condition* removes some of the noise from the sensor data and is computationally cheap, requiring only several algebraic operations for every sensor reading. In this wakeup condition the raw accelerometer data is passed through a low-pass filter based on an exponential moving average before being checked against the threshold. The two parameters that can be set for this wakeup condition are the threshold value and an alpha value between 0 and 1. The alpha value controls the "smoothness" of the filtered data (TODO: rephrase).

The *FFT LPF wakeup condition* is more accurate at removing noise from the sensor data. However, it is computationally expensive. In this wakeup condition the raw accelerometer data is passed through a low-pass filter based on Fast Fourier Transformations before being checked against the threshold. Two parameters can be set for this wakeup condition: the threshold value and a relative energy level value. The relative energy

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

value indicates the percentage of the original energy present in the raw acceleration that is to be removed during filtering.

4.2.2 Hardware Options

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

Our first option is a Texas Instruments MSP430 micro-controller. It has the advantage of being very low-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 five different continuous sensing methods: 1) always on, 2) duty cycling, 3) sensor data batching, 4) predefined activity detection, and 5) our filter-based approach for specifying custom wakeup conditions. 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 accelerometer readings by attaching a Google Nexus 4 to the back of the robot.

To get statistically significant results, the robot was programmed to perform its course multiple times. Each course generated a separate ground truth log and an accelerometer trace. In a course, the robot performed five different actions: standing idle, walking, seat-to-stand transition, stand-to-seat transition, and headbutt. To eliminate bias, these actions were generated randomly in each course. We divided courses into three different groups to cover a range 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: 90% for walking, 9% for transitions between seating and standing, and 1% for headbutts. This setups allows us to experiment with detecting actions that are common, somewhat frequent, and rare. Figure 6 shows the average amount of time that each action is being performed in the traces in each group, as a percentage of the average trace length. *TODO: mention how many courses from each group we ran*

5.2 Applications

We have implemented three applications:

- **Steps** counts how many steps the dog takes when it walks by looking for local maxima in the x-axis of the acceleration trace.
- **Stand-seat** detects transitions between seating and standing by monitoring changes in the acceleration due to gravity on the y and z axes.
- **Headbutt** detects a headbutt motion by searching for local minima on the y-axis of the acceleration trace.

Each application runs a set of low-pass filters based on fast-fourier transforms (FFT) on the main processor for activity detection. We use these applications for evaluating always-on operation, duty cycling, and

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

batching. The activity detection and wake-up condition sensing methods require programming the low-power processor. To evaluate these methods, we developed a variant of each application that runs three types of filters on the low-power processor: 1) simple threshold, 2) exponential moving average, and 3) FFT. The low power processor wakes up the phone when these filters indicate that an event of interest has been detected. The application then runs its FFT algorithms on the main processor to eliminate false alarms, i.e., events that where misidentified by the low-power processor.

5.3 Trace-based Simulator

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.

The simulator processes the accelerometer 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 these readings and the wake-up condition, the simulator determines when the device would be woken up. The simulator also determines what readings need to be sent to the running applications. At the end of the run, the simulator outputs the following metrics:

Might be better to put the following text in a table -Ashvin.

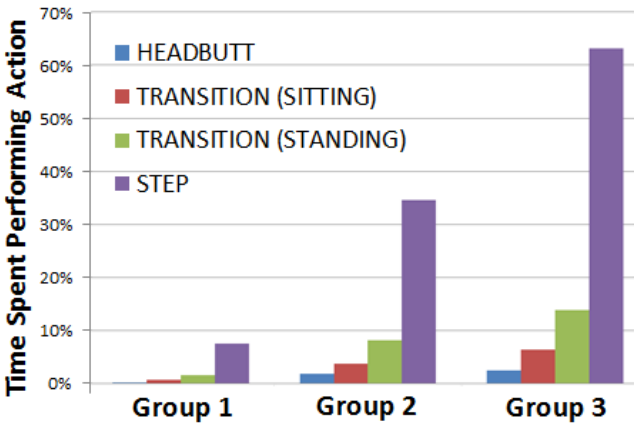

Figure 6: Time spent performing actions of specific types as a percentage or trace length

1. Amount of time the device would have been awake.
2. Total number of times the device was woken up.
3. Number of times that the device was woken up that did not result in an event of interest being detected by the application.
4. True positives, false positives and false negatives for each event of interest. **define each precisely -Ashvin**.
5. Recall and precision for each event of interest. **define each precisely -Ashvin**.
6. Power consumption over the duration of the trace. This value is derived from the energy measurements presented in Section 4, the amount of time the device is awake, the amount of time it is asleep, and the number of transitions between the two states.

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

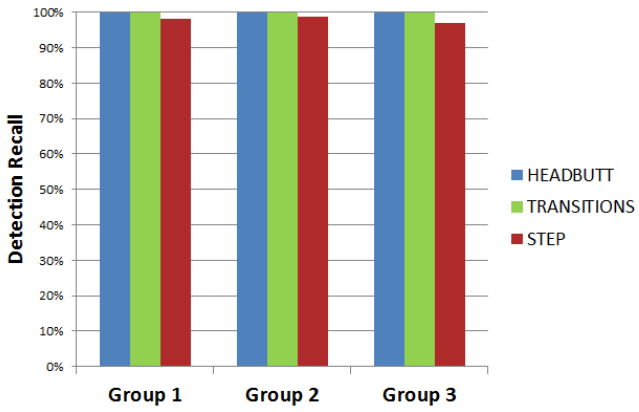

Figure 7: Always Awake: Recall by Group

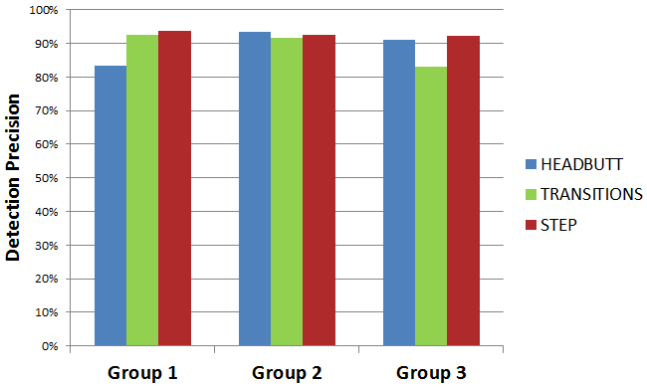

Figure 8: Always Awake: Detection Precision by Group

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.

I think the following text should go in the results section. -Ashvin

Additionally, the Always Awake simulations show the accuracy of the implemented applications in terms of event of interest detection precision and recall.

Figure 7 shows the average recall for all applications when executing on a phone that is kept always-on. With the exception of the headbutt action in Group 1, event recall is above 95% for all actions and groups. The headbutt action in Group 1 appears to be an outlier because Group 1 contains traces with the least amount of actions and headbutts are the scarcest actions, occurring only about once per trace. 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%.

6. RESULTS

	Action	Power (mW)	Recall
Group 1	Walking	42.2	98%
	Headbutts	42.2	100%
	Transitions	15.6	100%
Group 3	Walking	267	97%
	Headbutts	62.7	100%
	Transitions	31.1	100%

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

In this evaluation we will be answering 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 conditions does the wakeup condition approach improve performance (over the other approaches), and by how much?
4. What are the benefits of having different wake up conditions (filters)? Do different applications require different wake up conditions or is a static wake-up condition sufficient for all applications?
5. Are the best filter/threshold combinations dependent on the usage scenario?
6. How sensitive are the energy savings to the configuration thresholds used in the filter?

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 on 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 on scenario.

We note the following:

The duty cycling approach has significant power savings for sleep intervals greater than 5 seconds, compared to the always on scenario. However, in cases where the event of interests occurs rarely (group 1), the power savings of duty cycling are suboptimal because of numerous wake-up that result in no events of interest being detected. The greatest drawback of this approach is that it is not appropriate for applications that requires a high event of interest recall. Using a 5-second sleep interval, the highest achieved recall was 56

The batching approach achieves the same event of interest recalls as the always on scenario, and is able

	Approach	Sleep Interval (seconds)	Power (mW)	Recall		
				Walking	Headbutt	Transitions
Group 1	Always Awake		323	98%	100%	100%
	Batching	2	84.0			
		5	60.5			
		10	52.6			
		20	48.7			
		30	47.5			
	Duty Cycling	2	212	94%	57%	100%
		5	140	82%	14%	35%
		10	92.1	63%	29%	23%
		20	59.4	48%	14%	10%
		30	44.2	31%	7%	4%
Group 3	Always Awake		323	97%	100%	100%
	Batching	2	84.0			
		5	60.5			
		10	52.6			
		20	48.7			
		30	47.5			
	Duty Cycling	2	261	89%	47%	90%
		5	209	76%	28%	39%
		10	166	64%	27%	8%
		20	112	42%	19%	7%
		30	82.7	30%	9%	5%

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

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. 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.

The wake-up conditions approach has 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

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 2 applications.

We start off by selecting the best wake-up condition for step detection. We used this wake-up condition and attempted to detect headbutts and stance transitions. The top part Table 4 shows the results in terms or detection recall and power consumption. The bottom half of the table shows the results for the same metrics when we used the best wake-up condition for those specific actions. We note that in every single case, using a custom filter increased detection recall. For the headbutt action, recall was more than doubled for every group. For transitions, using a custom filter increased recall to 100

We can conclude that using a fixed wake-up condition (or a hardware supported action) may work very well for a small set of application, but is extremely inefficient for most other applications. As such, developers should

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%	62.7%	69.3%	87.5%	76.4%	78.3%
	Power	42.2	154	267	42.2	154	267
Best custom	Recall	100%	100%	100%	100%	100%	100%
	Power	42.2	62.4	62.7	15.7	27.2	31.1

Table 4: Detection recall and power consumption for the Headbutts and Transitions actions using fixed or custom wake-up conditions

be able to configure the wake-up conditions based on the needs of their application.

6.3 Usage Dependence

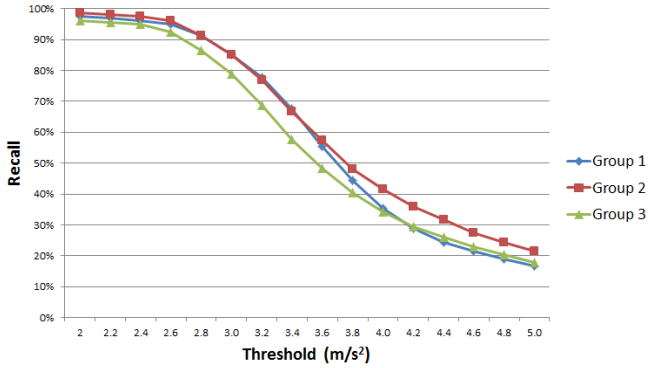

Figure 9:

This section answers question 5. Figure 9 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

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

6.4 Threshold Sensitivity

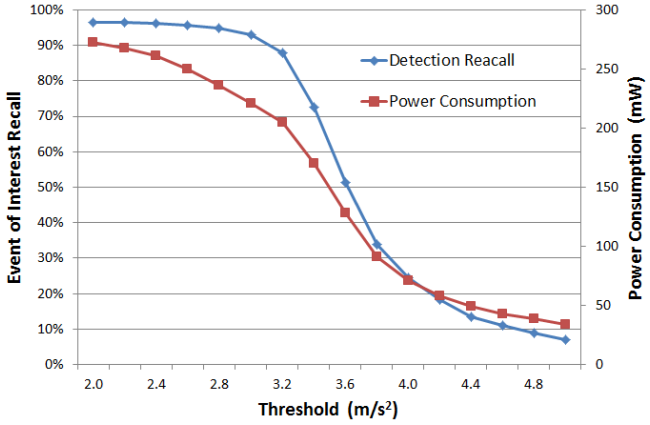

Figure 10:

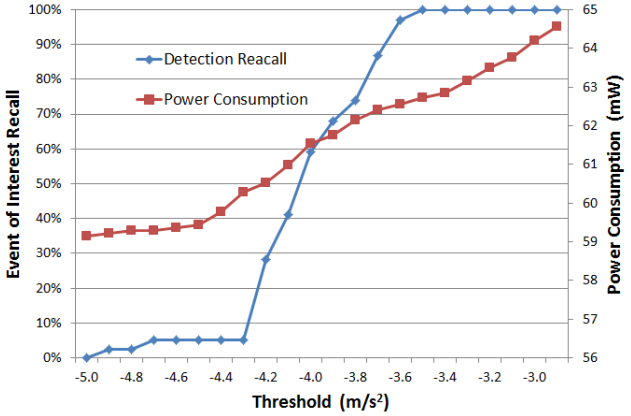

Figure 11:

This section answers question 6. Figures 10, 11 and **??** show the detection recall and power consumption for each application using the best wake-up condition for that specific application. From the figures we note that the choice of threshold is very 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.

TODO: the choice of threshold seems forgiving, but I'm not sure how to explain it

7. RELATED WORK

Code offloading — Odessa and MAUI

Mitigate sensing cost for acquisition of context attributes (such as AtHome and IsDriving) — Ace

Other approaches that save energy using asymmetric processors Turducken Reflex

8. REFERENCES

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

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

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

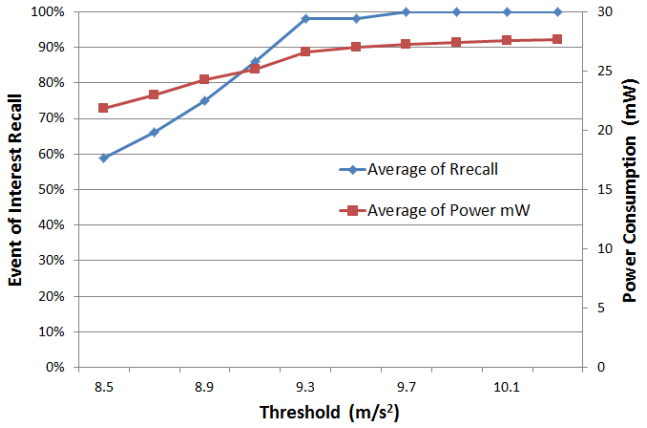

Figure 12:

[4] 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.

[5] 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.