SmartSensors: Energy-efficient sensing on mobile devices

1. INTRODUCTION

Smart phones and tablets are overwhelmingly used today 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 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 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 medical and health monitoring applications, such as pedometers and fall detectors, to participatory sensing applications, such as pollution and traffic monitoring. While the processing demands of these applications is modest most of the time, they require the 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 [1]. 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 prevent (so far) its adoption in commercial devices. Instead, smartphone manufacturers have recently incorporated low-power processors into their architectures, but have limited application developers to APIs that provide fixed functionality, including batching sensor readings and recognizing a small number of predefined activities that can be used as *wake up* conditions.

In this paper we argue that these APIs are insufficient in order to support 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. On the other hand, an API based on activity recognition 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, enabling programmers to chose among a set of predefined filters that implement commonly used algorithms (e.g., FFT, exponential average) with parameters that can be configured by the developer to meet application requirements. By providing access to lower-level filters, as oppose to higher-level activity detectors, our approach provides a better balance between flexibility and ease of deployment. By letting the developer select and configure the filtering algorithm used by the low-power hardware, our approach empowers developers to design a broad set of wake up conditions that can be used to detect a wide range of activities. This functionality could be leveraged by user-level activity recognition libraries that could then provide simple wake up conditions for a large number of activities.

To evaluate the benefits of our approach we 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 **??** present our experimental setup and the results from our evaluation. Finally, Sections 6 and **??** describe our work in the context of related work and conclude the paper.

2. BACKGROUND

Mobile application developers can choose between several approaches when attempting to perform continuous sensing. The availability of some of these approaches is dependent upon the mobile device's hardware and overlaying operating system.

The most basic approach that can be used on any sensor-enabled mobile device is to have the mobile device always on. On the Android operating system, the application would acquire a wake lock to prevent the device from sleeping and register 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. This will continue to happen until the listener is unregistered or until the wake lock is released. This approach has the potential to achieve the highest possible event of interest recall because all the sensor data is passed to the application(s). However, because the device does not experience any sleep periods, the power consumption of this approach is extremely high.

To avoid keeping the device awake for extended periods of time, a duty cycling approach can be used. In such an approach, the device is allowed to sleep for fixed, usually regular, periods of time. An application can implement duty cycling by scheduling an alarm using the system's alarm service and ensuring it does not have have a wake lock acquired. The application is then notified by the operating system when the sleep period ends. At this time, the application can collect sensor readings for a period of time, then reschedule another alarm. This approach has a couple of apparent drawbacks. First off, events of interest will not be detected if they occur while the device is sleeping as the application does not receive any sensor data that would have been produced at those times. Secondly, the device might wake-up to discover that the event of interest is not actually occurring. This approach may be slightly improved in terms of event of interest recall by dynamically adjusting the length of time readings are collected for. If after a wake-up the application detects an event of interest, it can continue to collect sensor readings until events of interest are no longer being detected, at which time it can return to its regular duty cycling schedule. Fundamentally, this approach is a trade-off between energy savings and event of interest recall. Staying awake for a larger proportion of the time to collect sensor data is expected to improve recall, but reduce the amount of energy savings. Finding a balance between power savings and event of interest recall is not trivial as it not only depends on the specific application, but also on the user. 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.

The newly released Google Nexus 5, running Android 4.4, introduces an extension to duty cycling referred to as sensor data batching. This approach requires hardware support and is currently not available on any devices other than the Nexus 5. In this approach, the device hardware can collect sensor readings while the main processor is sleeping. At wake-up, the entire batch of sensor data is delivered to the application(s) that registered a listener for that specific sensor. Batching has the great advantage that the applications receive all the sensor data, meaning that no events of interest will be missed. However, this comes at the cost of 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.

Multi-processor architectures allow computation offloading and wake-ups via interrupts. However, particular architectures may limit the degree of programmability of the peripheral processor.

In Refex[1], the authors show an architecture where the application developers can program their own sensor data analysis algorithms that are to be executed on the peripheral processor. Unfortunately, developers have to be exposed to the full complexity of the architecture, making application development more difficult. Even simple sensor-driven applications have to be re-factored into distributed programs. Additionally, it is unclear how the architecture can support multiple applications that necessitate data from the same sensor. The high level of programming complexity is very likely a cause why such an architecture is not supported by current commercial systems.

On the other side of the programming complexity spectrum, recent mobile devices limit computational offloading to a small, pre-defined set of activity detection algorithms. For example, the Motorola Moto X 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 part of the screen when the device is taken out of a pocket or start the camera application when two twists of the wrist are detected. Additionally, Android 4.4 allows future devices (if they contain the appropriate sensors) to wakeup on specific trigger events such as "significant motion" or a step. However, the application developers have no control over the underlying algorithms or customization of parameters. Fundamentally, this approach provides callbacks for pre-defined activities such as walking or specific phrase recognition. This approach works great if the application developer is interested in one of the pre-defined activities. However, the approach provides no support for applications that are interested in activities that are not part of the pre-defined set. The main advantage of this approach is it ease of use from the point of view of application developers, provided that the platform supports the activities of interest. Also, very good energy savings are achieved as the device only wakes up when the event of interest occurs.

3. APPROACH

Our aim is to enable low-power, continuous sensing tasks on mobile processors. As described earlier, the current approaches for low power sensing trade ease of programming with the generality of the approach. On the one hand, fully programmable offloading of code to low-power processors allows developing arbitrary applications [TODO:cite]. However, it raises several challenges, such as requiring a split or complete redevelopment of application functionality, and carefully choosing the code that can be offloaded so that it can run in real time. The latter is complicated because it depends on the type and functionality of the low-power processor that is available. Furthermore, since each application is written independently, this approach makes it hard to optimize power consumption for multiple sensing applications. Alternatively, the phone can be designed to support a predefined set of activities but this approach limits the number and types of applications that can be supported.

Our approach for low-power sensing takes a middle ground and provides a filter-based sensor API for application programs. Applications choose from a pre-specified set of event filters that are run by the low-power processor to detect application-specific events of interest. 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 combine the filtering and wake up operations when applications register interest in the same set of events.

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.

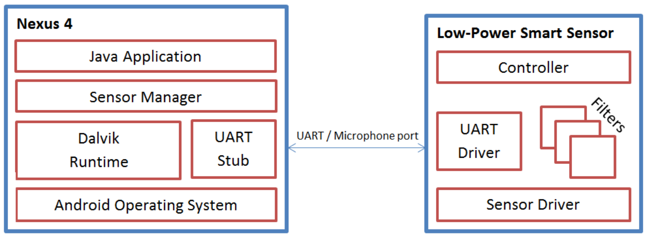

Figure 1: High-level architecture of our prototype implementation

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 1 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 2 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 3 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 simplest 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

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

Figure 2: 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 3: Usage of the SensorManager with a wakeup condition

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 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

To create a fair comparison of the various wake up approaches, it is necessary to ensure a high control of consistency and repeatability of the input sensor data. Unfortunately, this is very difficult to achieve during live experiments performed by human subjects. Instead, we opted to collect accelerometer traces and run trace-based simulations.

Additionally, our simulations requires the availability of ground truth in order to accurately compute metrics such as event of interest detection precision and recall. Annotating accelerometer traces collected from human subjects with ground truth information about the performed actions is both extremely time-consuming and prone to errors. Instead, we chose to use a robot to perform a set of actions and simultaneously keep a log of ground truth information about the performed actions.

The use of a robot could have enabled us to run live experiments because performing the same sequence of actions multiple times would result in similar accelerometer traces. We opted not to do so for several reasons. Each run takes upwards of an hour to execute. Our goal was to obtain results for a multitude of wake-up approaches, each under multiple parameter configurations. Additionally, we wanted to reduce the impact of outliers on our results by running each experiment on a multitude of robot action sequences. The combination of all action sequences and wake-up configurations would have required over a year of continuous live experimentation. Running trace-based simulations allows us to perform an exhaustive exploration of the parameter configuration space. Moreover, taking fine grain power consumption measurements while the robot is in motion is not trivial. Nonetheless, this is part of our future work.

The use of a robot significantly limits the set of actions that can be performed in our experiments. Since we are mainly interested in actions that would normally be performed by humans, we tried to configure the robot to perform actions with similar acceleration signatures. We decided upon a set of 4 actions: walking, headbutts and two stance transitions. Robot walking has a similar acceleration signature as its human counterpart, though at a much 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 as humans sitting down and standing up.

To collect the accelerometer traces for our experiments we used an AIBO ERA 210 robot (Figure 4). The robot would perform a sequence of actions and record timestamps for the beginning and end of each action.

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

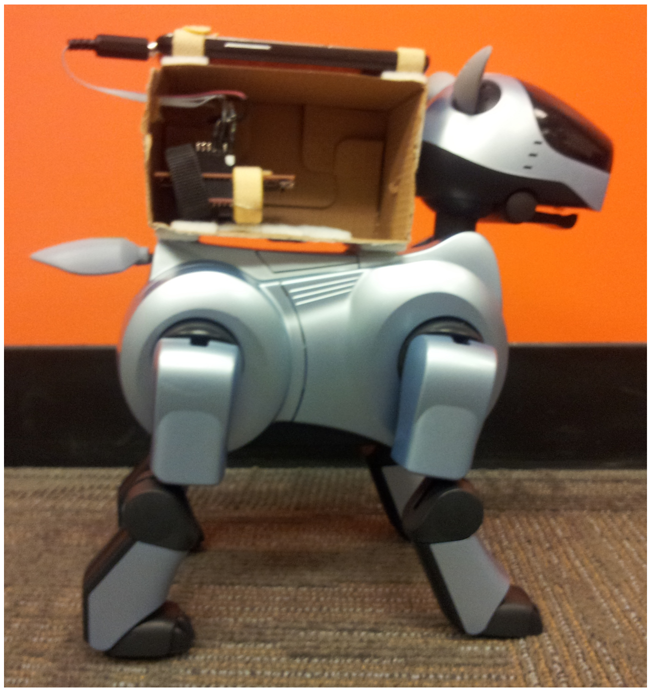

Figure 4: AIBO ERS 210 robot used to collect the accelerometer traces

The set of performed actions and timestamps were outputted to a log file that can be used by the simulator as ground truth. To record the accelerometer trace, we attached a Google Nexus 4 to the back of the dog and ran an application that collected all the accelerometer readings for the duration of the circuit.

The level of activity can vary significantly between mobile device users. For example, the mobile device of an individual working in an office environment may experience low levels of motion (e.g. sitting on a desk for extended periods of time), while the mobile device of someone constantly in motion (e.g. bike courier) may only experience short period at rest. In our experiments we wanted to cover a wide range of activity level. We divided our traces into 3 groups based on activity level. Groups 1, 2 and 3 contain traces in which the robot performs actions approximately 10%, 50% and 90% of the time, respectively. Within each trace, we also wanted the amount of time each action type is being performed to vary considerably. Figure 5 shows the

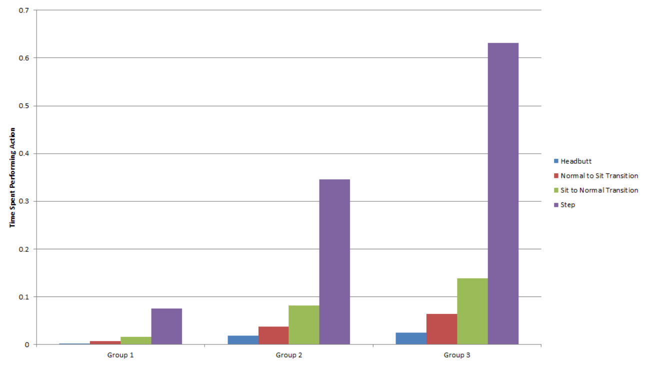

Figure 5: Time spent performing actions of specific types as a percentage or trace length

average amount of time each action is being perform for the traces in each group, as a percentage of the average trace length.

We implemented 3 applications that attempt to detect the 4 different actions from the accelerometer readings in each trace. A first application looks for local maxima within a certain range in the x-axis acceleration to detect steps taken by the robot. Similarly, the second application looks for local minima within a range in the y-axis acceleration to detect headbutt actions. The third application monitors changes in the acceleration due to gravity on the y and z axes in order to detect stance transitions between a normal posture and sitting. All 3 applications pass the accelerometer data they are interested in through low-pass filters based on Fast-Fourier Transformations.

5.1 Metrics

In our evaluation we are investigating several metrics:

1. the power consumption of the system
2. the recall of the event of interest
3. the precision of event of interest detection
4. the time delay incurred before the application becomes aware that an event of interest has occurred
5. the precision of device wake-ups

Recall is the fraction of events of interest that have been successfully detected. We calculate event of interest recall by dividing the number of true positives by the total number of performed events of interest as indicated by the ground truth data. Precision is the fraction of the detected events of interest that are present in the ground truth. For a particular event of interest we calculate precision by dividing the number of true positives by the number of detected events of interest of that type. Device wake-up precision is a measure of how often the device unnecessarily wakes up (i.e. wakes up and does not detect an event of interest).

5.2 Trace-based Simulator

Each collected accelerometer trace 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.

For each run, the simulator takes the following inputs:

1. a parametarized wake-up approach
2. a file containing the accelerometer readings, sorted by timestamp
3. a file containing information about ground truth

The simulator processes the readings one at a time, similarly to how a mobile device would receive accelerometer readings in real-time, as they are produced by the sensor. Based on the wake-up approach, the simulator determines which of the readings would cause the device to wake-up. Also, the simulator determines what readings need to be sent to the running application(s). At the end of the run, the simulator outputs relevant metrics such as:

1. amount of time the device would have been awake
2. total number of device wake-up
3. number of wake-ups that did not result in an event of interest being detected
4. true positives, false positives and false negatives for each event of interest
5. recall and precision for each event of interest

5.3 Power Consumption Model

To complement our simulator, we created a model to estimate the power consumption for each of our simulator runs. The model is based on real power measurements presented in section 4. The inputs of the model are the amount of time the device is awake, the amount of time it is asleep and the number of transitions between the two states. Based on these values, the energy model outputs the average power consumption over the duration of the trace.

5.4 Baseline

In the Always Awake approach, the device is presumed to be awake for the entire duration of the trace and all of the collected accelerometer readings are passed

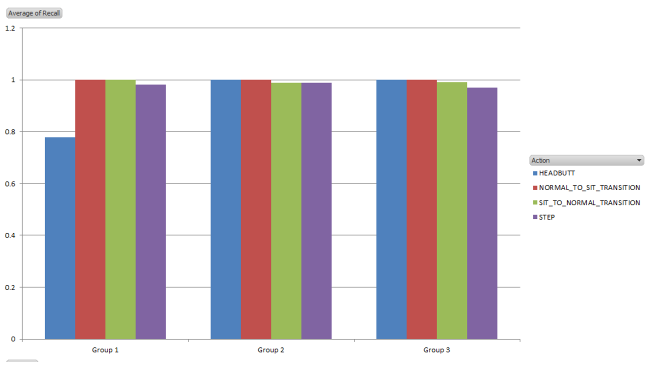

Figure 6: Always Awake: Recall by Group

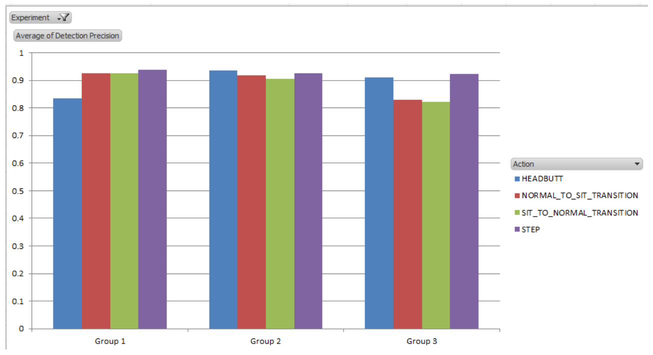

Figure 7: Always Awake: Detection Precision by Group

to the applications. We are using the Always Awake approach as a baseline to compare all the following wake-up approaches. Additionally, the Always Awake simulations show the accuracy of the implemented applications in terms of event of interest detection precision and recall. Figure 6 shows the recall of each of the events of interest averaged over the traces in each group. We note that with the exception of the headbutt action in Group 1, the event of interest 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%.

The power consumption using this approach is equivalent to the power consumed by the device in the awake state. In the case of the Nexus 4 used in our experiments, the power consumption is 323 mW.

6. 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

7. REFERENCES

[1] 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. EXTRA FIGURES

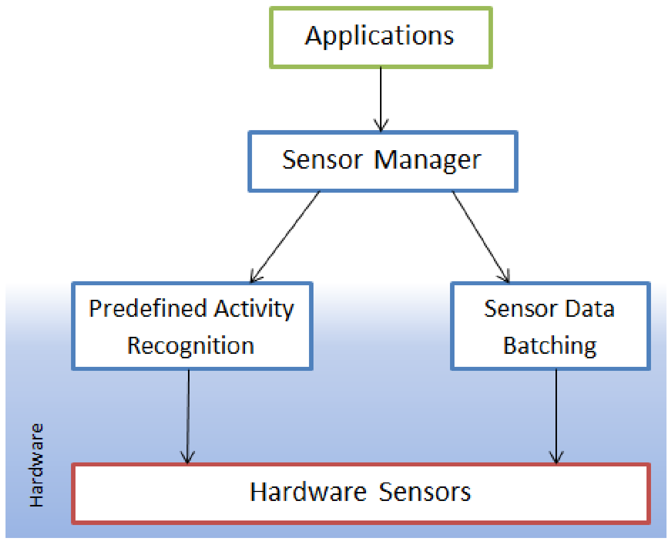

Figure 8: Simplified architecture of Android 4.4 sensor system

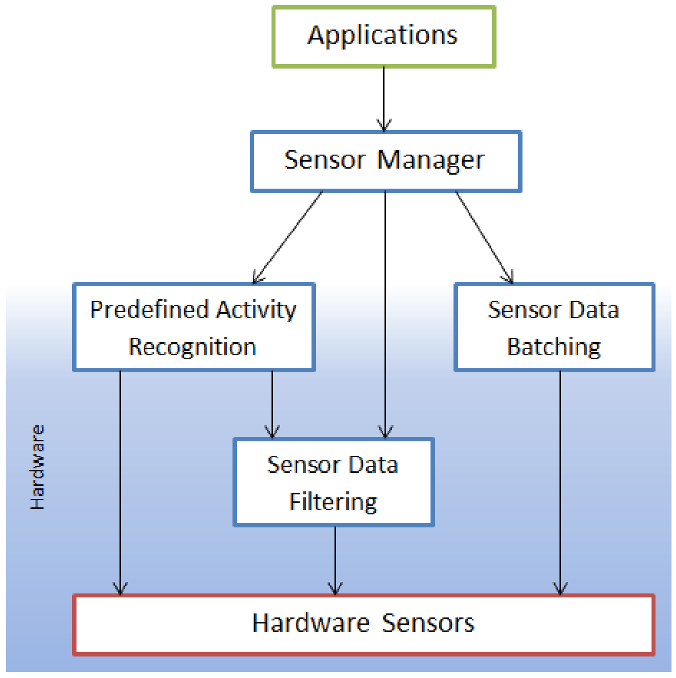

Figure 9: Proposed architecture based on Android 4.4 sensor system