SmartSensors: Energy-efficient sensing on mobile devices

1. 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.

2. PROTOTYPE

We chose to focus our efforts on accelerometer-based applications because there exists a wide-range of such applications available on the various mobile application markets. We considered applications that use the device's camera or microphone, as they are also fairly common. However, a scenario where multiple applications are using the camera or the microphone concurrently seemed fairly unrealistic.

2.1 Sensor Data Filters

We implemented three types of sensor data filters (or wakeup conditions) of varying complexity. The simplest filter is based on thresholding over raw accelerometer readings. Such a filter is satisfied if the detected acceleration exceeds the threshold. However, we noticed that accelerometer data is particularly noisy. Our expectation was that a simple thresholding filter would be satisfied more frequently than necessary because of the noise present in the sensor data, potentially causing unnecessary device wakeups. Our second and third filters are based on threshold on data that has been passed through a low-pass filter in order to remove some of the noise (TODO: I think this will be confusing. I am using the word filter multiple times with different meanings). The second filter uses a computationally cheap low-pass filter based on an exponential moving average. The third filter uses a more accurate and computationally expensive low-pass filter based on Fast Fourier Transformations.

2.2 Software Changes

This subsection provides an overview of the API that would be available to application developers and a description of the changes that needed to be made to the operating system.

2.2.1 API

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 1 shows a code example of a typical application registering a listener with the Android SensorManager for accelerometer readings.

We would modify the SensorManager to include functions that take an additional parameter that specifies which wake-up condition to be used. The available types of wakeup conditions would be predefined implementation of a WakeUpCondition interface. Figure 2 shows an example of the modified code to include a wakeup condition that is satisfied when the acceleration on the x-axis exceeds $12m/s^2$.

2.2.2 Changes to OS

TODO: I'm not confident about describing these changes

2.3 Hardware

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, but it has limited memory and cannot perform complex analysis of sensor data in real-time. In our preliminary 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 microcontroller powered by a Cortex-M4 CPU. It can batch a higher number of accelerometer readings and perform more complex data analysis such as FFT-based low-pass filtering in real time, but it has a greater energy footprint.

TODO:Communication with Nexus 4. Use of UART over mic port to wake up device (I'm not sure how this

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

Figure 1: 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 2: Usage of the SensorManager with a wakeup condition

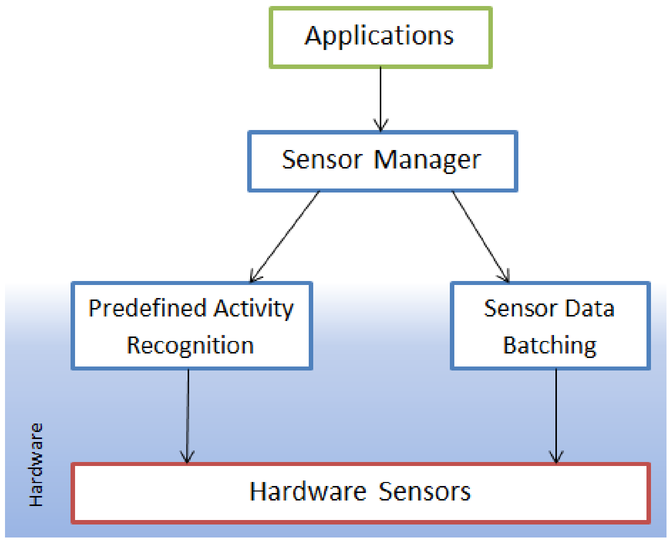

Figure 3: Simplified architecture of Android 4.4 sensor system

works)

2.4 Power Profiles

We performed power measurements on a Google Nexus 4 running Android 4.2.2. 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 results, we decided to run all the power measurements with GSM, WiFi and GPS turned off.

The power measurements for both micro-controllers are also presented in Table 1 on page 4.

3. RELATED WORK

Code offloading — Odessa and MAUI

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

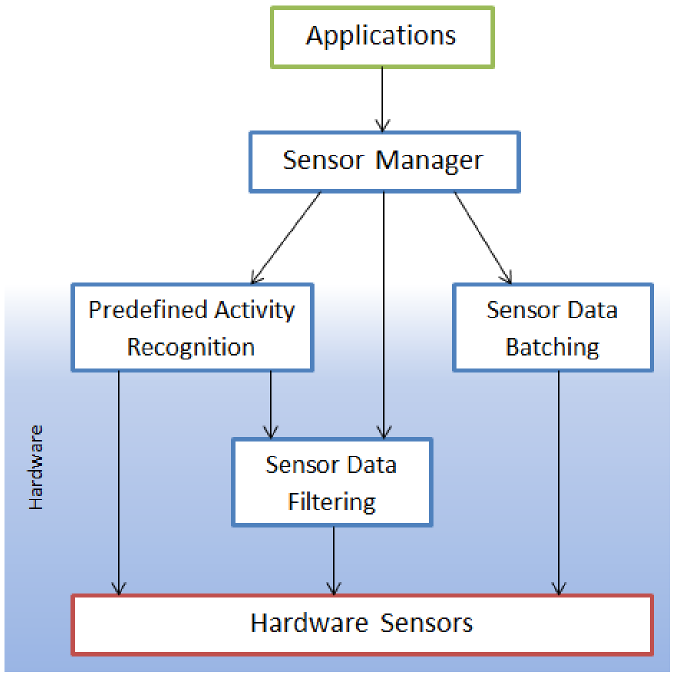

Figure 4: Proposed architecture based on Android 4.4 sensor system

Other approaches that save energy using asymmetric processors Turducken Reflex

4. REFERENCES

[1] X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them.

Device	State	Average Power Consumption (mW)	Average Duration
TI MSP430	Awake	3.6	N/A
TI Stellaris	Awake	49.4	N/A
Nexus 4	Awake, running a pedometer application with data from the internal accelerometer	323	N/A
Nexus 4	Asleep	9.7	N/A
Nexus 4	Asleep-to-Awake Transition	384	1 second
Nexus 4	Awake-to-Asleep Transition	341	1 second

Table 1: Power Measurements