Sidewinder: An Energy Efficient and Developer Friendly Heterogeneous Architecture for Continuous Mobile Sensing

Daniyal Liaqat
University of Toronto
dliaqat@cs.toronto.edu

Silviu Jingoi
University of Toronto
silviu@cs.toronto.edu

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Wilson To
University of Toronto
wilson.to@mail.utoronto.ca

Kevin Lee
University of Toronto
kevinalexander.lee@mail.utoronto.ca

Italo De Moraes Garcia
University of Toronto
italo.demoraesgarcia@mail.utoronto.ca

Manuel Saldana
Huawei
manuel.saldana@huawei.com

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 noise pollution, traffic and seismic activity 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, resulting in poor battery life. This paper presents Sidewinder, a new approach towards offloading sensor data processing to a low-power processor and waking up the main processor when events of interest occur. This approach differs from other heterogeneous architectures in that developers are presented with a programming interface that lets them construct application specific wake-up conditions by linking together and parameterizing predefined sensor data processing algorithms. Our experiments indicate performance that is comparable to approaches that provide fully programmable offloading, but do so with a much simpler programming interface that facilitates deployment and portability.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Heterogeneous (hybrid) systems

Keywords Mobile Computing, Continuous Sensing, Energy Efficiency, Heterogeneous Architecture

1. Introduction

Modern smartphones are fitted with a wide assortment of sensors. A typical smartphone, such as the LG Nexus 5, contains an accelerometer, barometer, compass, gyroscope, proximity sensor, ambient light sensor, hall effect sensor, geo-spatial positioning sensors (GPS, GLONASS, Beidou), a microphone and two cameras. Despite having such a rich array of sensors, most of them go unused most of the time. The way smartphones are typically used is that the user will open an application such as a game, news or social media application, interact for a few minutes and then put their device to sleep. Taking advantage of this usage pattern, mobile processors have been heavily optimized to rely on a low-power sleep state when not in use in order to improve battery life.

While this approach has fared well with how devices are currently used, it falls short in the face of emerging continuous sensing applications, examples of which range from context-aware applications [11, 16], such as medical applications that improve our well-being [15, 26, 32] to applications that use participatory sensing to get a better understanding of the physical world, such as noise pollution monitoring [21, 22], traffic prediction [17] or earthquake early warning [23]. While the processing demands of these applications are modest most of the time, they require continuous collection of sensor data, which prevents the processor from entering its' low-power sleep state, resulting in poor battery life and ultimately, a slow emergence of continuous sensing applications.

It is generally accepted that the solution to this problem lies in a heterogeneous architecture where one or more low-power, peripheral processors (known as a sensor hub) collect and process sensor data while the main processor is in its sleep state. When the low-power sensor hub detects an event of interest, it wakes up the main processor, allowing for further processing of sensor data. The programming model for this heterogeneous architectures remains, however, an open research question. On one end of the spectrum, researchers have proposed a **fully programmable model** [19, 27, 31] that allows application developers to write arbitrary code that runs on the low-power processor. While this would provide potentially great flexibility and power savings, there are many drawbacks. Application developers would not only have to be familiar with programming the low-power processor, they would have to account for hardware differences between devices. It is also unclear how this model would support different applications that

are running concurrently. On the other end is **predefined activity**, an approach where hardware manufacturers provide a low-power processor which is hardwired to detect a few specific events. Both Apple and Android provide frameworks for detecting predefined activities such as significant motion and steps [2, 3]. Another example is the Motorola Moto X smartphone [7, 10] that has a dedicated Natural Language Processor which is used to wake up the device when the user says a certain phrase such as "OK Google Now". Such frameworks are very easy to use from a developer's point of view and provide significant energy savings, but they are very limited because they only allow detection of events that have been pre-programmed into the device by the manufacturer.

This paper presents *Sidewinder*, a new approach for continuous mobile sensing that splits the work of energy-efficient event detection between the platform and the application developer. With Sidewinder, the platform implements common sensor data processing algorithms (e.g., windowing, noise reduction, feature extraction, admission control) that execute on a low-power sensor hub, and application developers construct custom wake-up conditions by linking together and parameterizing these sensor data processing algorithms. The custom wake-up conditions execute on the low-power sensor hub and, when events of interest are detected, the main processor is woken up and the rest of the application code is invoked.

While heterogeneous architectures have been used previously, the innovation in Sidewinder is the collaborative approach where the platform provides sensor processing algorithms and developers parameterize and chain these algorithms together to create wake-up conditions. Processing algorithms are written natively for the low-power sensor hub so application developers do not need to worry about writing code for the underlying heterogeneous architecture.

To evaluate the benefits of Sidewinder, we developed applications that use accelerometer readings or audio data to detect several events of interest. Additionally, we built a prototype implementation that extends a Nexus 4 phone with a low-power sensor hub. To enable us to conduct controlled and repeatable experiments, we mounted our prototype on a robot. Simulations conducted on accelerometer and audio traces collected in various environments show that Sidewinder can reduce the average energy required to run continuous sensing applications by up to 96% compared to keeping the phone awake at all times, while matching the detection recall and precision of the always-on approach. Moreover, for most of our usage scenarios, Sidewinder achieves over 90% of the power savings achieved by a "perfect" wake-up mechanism, indicating that an implementation that supports custom code offloading will achieve only marginal additional improvements.

The rest of this paper is organized as follows. Section 2 describes the Design of Sidewinder and 3 describes how we implemented the design. Sections 4 and 5 present our evaluation method and results. Finally, Sections 6 and 7 describe our work in the context of related work and conclude the paper.

2. Design

Continuous mobile sensing approaches have to address two main constraints: maximizing detection accuracy and minimizing energy consumption. Users expect high precision and recall and user experience is adversely affected when the application misses or over-reports events of interest. Achieving both high recall and precision requires highly specialized algorithms tuned to the event of interest. Specifically, our experience suggests that getting the last few percentage points in precision and recall is difficult and requires complex algorithms and fine parameter tuning. Additionally, these complex algorithms typically run on a fully featured processor, which is detrimental to battery life since a large portion of energy savings comes from keeping the main processor in a sleep state.
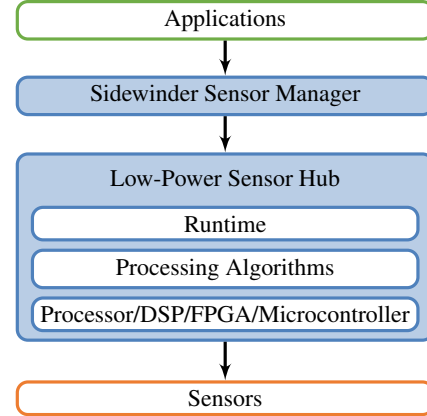

Figure 1: Proposed system architecture. The sensor manager is part of the OS, and the Sensor Hub and Sensors are hardware provided by the manufacturer

We achieve our two main goals (energy efficiency and high precision/recall) by using a multi-stage pipeline of algorithms to create a more complex classifier. Earlier stages are used to achieve the majority of available energy savings and later stages can then optimize for high detection precision and recall. For example, a voice recognition algorithm may have three stages, the first stage determines if microphone data contains sound, the second stage determines if the sound is human speech and the third stage converts the speech to text. The first two stages are relatively simple and can be run on a less powerful, more energy efficient processor. They have high recall but may have low precision (i.e. they will let through a high proportion of events that contain speech, but not all events passed through will contain speech). Since the first two stages are run on a low-power processor and reduce the amount of time the main processor has to be awake, this will result in an overall saving of energy.

Based on this observation, Sidewinder encourages continuous mobile sensing applications to be structured as pipelines of processing algorithms of increasing complexity: simple yet high recall, moderate precision algorithms that run continuously on a low-power sensor hub, providing energy efficient wake-up mechanisms for higher complexity algorithms that run on the main CPU and provide both high recall and high precision.

To facilitate the creation of wake-up conditions capable of running on a low-power sensor hub, Sidewinder provides a set of commonly used sensor data processing algorithms that are ready to run on the sensor hub. These algorithms can be parameterized and chained together to create wake-up conditions.

We conjecture: 1) that it is possible to implement custom wake-up conditions for a wide range of applications by configuring a small set of common processing algorithms; and 2) that this approach will achieve comparable energy savings to an alternative implementation that supports full programmability.

2.1 Sidewinder

Sidewinder is a new approach for continuous mobile sensing that divides the responsibility of energy-efficient event detection between the manufacturer and the application developer. The manufacturer provides a low-power sensor hub and implements common sensor data processing algorithms that execute on the sensor hub. Applications construct custom wake-up conditions for events of interest in their application code. These wake-up conditions are then pushed to and executed continuously on the low-power sensor

hub and, when events of interest are detected, the main processor is woken up and the application code is notified.

Figure 1 shows the architecture of a system that uses Sidewinder. Applications interact with a sensor manager to define a custom wake-up condition. The manager contains an API for parameterizing and chaining algorithms that are available on the low-power sensor hub. Developers can use this API to create their wake-up conditions. The low-power sensor hub contains commonly used algorithms for windowing, filtering, transformations, feature extraction and admission control. Once configured by the developer, the wake-up condition is converted into an intermediate language by the sensor manager and pushed to a runtime or interpreter on the low-power sensor hub. When running on the sensor hub, the custom wake-up condition wakes the main CPU if an event of interest occurs.

We next describe the components of Sidewinder based on whether their implementation is the responsibility of the device manufacturer, the operating system, or the application developer.

2.1.1 The Manufacturer

The manufacturer is responsible for providing the hardware and software of the low-power sensor hub. The hardware could be a network of one or more processors, Digital Signal Processors (DSP), FPGAs or microcontrollers. For example, there could be one larger processor to handle all sensors and algorithms or a DSP for the microphone and an FPGA for each of the other sensors. The manufacturer also needs to provide a runtime to manage this hardware and an implementation of common sensor data processing algorithms. The runtime needs to be able to receive wake-up condition configurations from applications, configure the hardware and algorithms, execute the resulting wake-up condition and notify applications when an event of interest occurs.

The runtime could use an interpreter approach where it executes each algorithm, a compiler approach where it generates an executable or it could reconfigure FPGAs according to the requirements of the wake-up condition and the hardware available. In the case of FPGAs the algorithms will most likely be pre-compiled and the runtime would need to reconfigure according to the specific configuration.

The runtime provides a decoupling layer between the mobile platform (Android, iOS, etc.) and the hardware since the runtime is responsible for managing the hardware. And since the manufacturer will be providing both the runtime and hardware, any architecture for the low-power sensor hub can be used.

2.1.2 The Application Developer

The application developer creates custom wake-up conditions for their event(s) of interest. One important aspect of Sidewinder is that common sensor data processing algorithms are provided to the developer by the platform. This means for example, if the developer needs to use an FFT, they do not need to implement it themselves or find a library. Instead, they would use the system API to create a wake-up condition that uses an FFT. The API allows developers to parametrize the FFT and if needed, chain it with other algorithms together to create more advanced wake-up conditions. They can then use the API to push their wake-up condition to the low-power sensor hub.

Because wake-up conditions are defined by configuring generic algorithms designed to support a large set of applications, as opposed to writing custom code specific to any application, their performance may be suboptimal by design. To ensure that user experience is not adversely affected when the application misses events of interest, application developers should create conservative wake-up conditions that provide for high recall at the expense of lower precision. This approach will ensure that no events of interest are overlooked, but will result in some unnecessary wake-ups, i.e., false positives. Therefore, to ensure that the application does not adversely affect user experience by over reporting, additional filtering needs to be executed on the main processor on a wake-up event to eliminate any false positives. In Section 5, we show that while the moderate precision of wake-up conditions does result in additional energy use, the approach is nevertheless able to achieve 90% of the power savings achieved by an "ideal" wake-up mechanism.

2.1.3 The Operating System

The operating system needs to provide the API (part of the Sensor Manager in Figure 1) that allows developers to create wake-up conditions and push/receive data to/from the low-power sensor hub. A wake-up condition is pushed to the sensor hub in the form of an intermediate code to decouple the platform from the hub. The operating system will also need to provide a driver to allow communication with the hardware. Depending on the mobile platform (Android, iOS, Windows 10 Mobile), it is likely that the manufacturer will provide the driver.

2.2 Advantages

Sidewinder has many benefits:

Lower programming complexity. Programming complexity is decreased because application developers can use the predefined processing algorithms, rather than implementing their own. The intermediate language makes Sidewinder language independent so that developers can write their classifier in the same language as their application.

Better optimization. The hardware and software of the low-power processor is implemented by the manufacturer, allowing for much greater optimization by experts.

Better security. Providing access to these algorithms via an API has significant security advantages over the fully programmable offloading approach because application developers cannot execute arbitrary code on the low-power processor.

Improved portability. Programmers do not have to be aware of the specifics of the underlying hardware, nor create a version for every type of platform. Manufacturers are free to use any type of hardware they want (processor, DSP, FPGA or networks of processors/DSPs/FPGAs) as long as it can interpret and execute the intermediate language.

3. Implementation

In this section we outline our implementation of the components mentioned in the design section. We also describe the applications we developed to evaluate Sidewinder. We implemented Sidewinder on the Android platform. It is important to note that there are many different valid implementations of our design, ours is just an example of a valid implementation.

3.1 Sensor Manager

The Sidewinder sensor manager is based off the Android sensor manager [8]. It contains information about the available sensors and processing algorithms and gives developers access to them via the API.

3.2 API

The API allows developers to create wake-up conditions. It contains four major components:

- $ProcessingPipeline$. This represents the entire wake-up condition from the input sensors to the final output. The pipeline consists of one or more processing branches.

- $ProcessingBranch$. Branches represent the flow of data from either a sensor to an algorithm or between two algorithms. At

```java
public class SignificantMotion implements SensorEventListener {

    public void createClassifier() {
        ProcessingPipeline significantMotion = new ProcessingPipeline();

        ProcessingBranch[] branches = new ProcessingBranch[3];
        branches[0] = new ProcessingBranch(SidewinderSensorManager.ACCELEROMETER_X);
        branches[1] = new ProcessingBranch(SidewinderSensorManager.ACCELEROMETER_Y);
        branches[2] = new ProcessingBranch(SidewinderSensorManager.ACCELEROMETER_Z);

        branches[0].add(new MovingAverage(10));
        branches[1].add(new MovingAverage(10));
        branches[2].add(new MovingAverage(10));

        Algorithm vm = new VectorMagnitude();
        Algorithm ac = new MinThreshold(15);

        significantMotion.add(branches);
        significantMotion.add(vm);
        significantMotion.add(ac);

        SidewinderSensorManager sManager = (SidewinderSensorManager) getSystemService(
                SENSOR_SERVICE);
        sManager.push(significantMotion, this);
    }

    @Override
    public void OnSensorEvent(SensorData data) {
        // Do something with data
    }
}
```
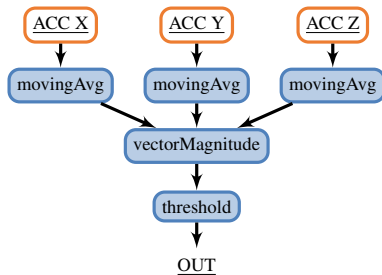
(a) Java representation

(b) Conceptual representation

```
ACC_X -> movingAvg(id=1, params={10});
ACC_Y -> movingAvg(id=2, params={10});
ACC_Z -> movingAvg(id=3, params={10});
1,2,3 -> vectorMagnitude(id=4);
4 -> minThreshold(id=5, params={15});
5 -> OUT;
```

(c) Intermediate representation

Figure 2: Various representations of a Significant motion pipeline

the start of the classifier pipeline, there may be any number of branches, each receiving data from any of the available sensor channels. At the end of the pipeline, there must be only one branch remaining. This means that if the pipeline contains multiple branches, aggregation algorithms need to be used to reduce the number of branches until a single branch is left.

- *Algorithm.* An algorithm is some operation that accepts one or more branches and produces one branch. For example, a moving average or admission control (threshold) algorithm accepts one branch and produces one branch. A vector magnitude algorithm accepts one or more branches and produces one. At the API level, these algorithms are simply stubs that represent the algorithm implantations at the low-power processor level.

- *Sensor Event Listener.* This is the Android SensorEventListener. It is a callback method that is registered with the sensor

manager that will be called when the custom wake-up condition is satisfied.

An example of a significant motion wake-up condition is given in Figure 2a. A conceptual diagram of the condition is given in Figure 2b. First a ProcessingPipeline object is created. Next, three branches, one for each axis of the accelerometer are created and each branch is given one axis as its source. Then, three MovingAverage algorithms are created, each with window size of 10, and one MovingAverage is added to each of the branches. A VectorMagnitude object and MinThreshold object are also created. The order in which these algorithms and branches are added to the ProcessingPipeline specify how they are chained together. Since the MinThreshold is the last algorithm in the pipeline, if it produces any result, the callback method will be invoked. Now that the pipeline

is configured, it, along with a SensorEventListener, is pushed to the SidewinderSensorManager.

3.3 Intermediate language

The intermediate language allows decoupling between the sensor manager and the low-power processor implementation. Upon receiving a wake-up condition configuration, the sensor manager generates its associated intermediate code. The intermediate code for the significant motion wake-up condition is shown in Figure 2c. Having the intermediate code allows developers to write their conditions in the same language as their application. In the intermediate code, each algorithm has a unique ID (generated by the sensor manager). In the example, the moving average algorithms are given IDs from 1, 2 and 3. Then the vector magnitude is setup to receive data from algorithms 1, 2 and 3 and the result of the vector magnitude is given to the admission control algorithm. Finally, the admission control algorithm is fed to OUT. A value being sent to OUT indicates that an event of interest has occurred and the main processor should be woken up.

3.4 Hardware

Our prototype is built around a Google Nexus 4 phone running Android 4.2.2. Since the Nexus 4 does not have an easily programmable sensor hub built in, we implemented our low-power sensor hub using a Texas Instruments (TI) MSP430 or LM4F120 microcontroller attached to an accelerometer sensor and a microphone. We chose to focus our efforts on these sensors because in our experience they are the most commonly used.

The Nexus 4 and microcontroller communicate over the UART [9] port made available by the Nexus 4 debugging interface via the audio interface jack. The serial connection provides sufficient bandwidth to support low bit-rate sensors, such as the accelerometer, a microphone or GPS. 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 [6].

3.5 Runtime

The main responsibility of the Sidewinder runtime is to execute the intermediate language. In this regard, the implementation of the runtime is very flexible. Our implementation of the runtime resembles a simple interpreter (written in C). It contains implementations of algorithms and a list of all available algorithms. Upon receiving a new configuration, the runtime allocates memory for each algorithm in the configuration. The interpreter then waits for sensor data to be available and feeds the data into the appropriate algorithm. If the algorithm produces a result, it sets a flag. The interpreter checks the flag and if necessary sends the result to the next algorithm. The flag is required because some algorithms may not always produce a result. A moving average with a window size of N will not produce a result until it has received N data points and a threshold will only produce a result when the threshold is met. The final algorithm feeds into OUT, indicating that the main processor should be woken up.

3.6 Processing Algorithms

Our algorithms are written in C and packaged with the runtime. Each algorithm operates on its own instance of a data structure. The data structure is created by the runtime and stores the algorithm ID, type, size, data, whether a result is available and the result. It can also contain any other data needed by the algorithm. Each time the algorithm needs to be run, the interpreter invokes the algorithm and passes it its data structure. The algorithm operates on the data available in the structure and, if required, stores the result in the structure and sets the hasResult flag. We implemented the following algorithms:

- **Windowing** Partitioning sensor data into rectangular or Hamming windows.
- **Transform**
  - Fast Fourier Transform (FFT) from time-domain to frequency-domain
  - Inverse Fast Fourier Transform (IFFT) from frequency-domain to time-domain.
- **Data Filtering**
  - Noise-reduction algorithms such as a moving average and exponential moving average.
  - FFT-based low-pass filtering.
  - FFT-based high-pass filtering.
- **Feature Extraction**
  - Magnitude of acceleration vector computation.
  - Zero Crossing Rate computation.
  - A set of statistical functions.
  - Determination of magnitude of dominant frequency.
- **Admission Control** Configurable high or low thresholds.

3.7 Applications

We developed six applications to run on the mobile device and, for each of the applications, we constructed a wake-up condition using the algorithms presented in Section 3.6.

3.7.1 Accelerometer Applications

We decided to use a robotic dog to conduct controlled and repeatable experiments. We developed three applications that detect activities that an AIBO ERA 210 robot can perform: walking, posture transitions, and headbutts. We chose these actions because they have similar acceleration signatures to human activities. It should be noted that, although classifiers to detect these activities will be similar (i.e. similar sensors used, similar algorithms and order of algorithms), each classifier requires its algorithms be parametrized specifically to the activity. This reinforces that a small set of algorithms can be used for numerous classifiers based on how they are chained and parameterized.

A walking robot has a similar acceleration signature as a human, though at a lower intensity. The headbutts are meant to represent very infrequent human actions such as falling. We found that robot stance transitions between the normal and sitting postures are very similar in their acceleration signature to humans sitting down and standing up. In Section 5 we show that the energy saving measured in our experiments with the robot approximate closely the results of experiments conducted on limited traces collected from human subjects.

Steps Counts how many steps the robot takes when it walks. The algorithm is based on the human step detection algorithm proposed by Ryan Libby [18]. 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.

Transitions Detects transitions between sitting and standing. The application monitors changes in acceleration due to gravity on the y and z axes to determine the orientation of the device. If the z-axis (up-down relative to the dog) acceleration is between $9m/s^2$ and $11m/s^2$, and the acceleration on the y-axis (front-back relative to the dog) is between $-1m/s^2$ and $1m/s^2$, the device is in a horizontal position and the robot is assumed to be in a standing posture. Similarly, if the z-axis acceleration is

Steps

3-Axis Accelerometer

Feature Extraction
x = x-axis Acceleration

Noise Reduction

Admission Control
threshold <= x <= threshold

Headbutt

3-Axis Accelerometer

Feature Extraction
y = y-axis Acceleration

Noise Reduction

Admission Control
y <= threshold

Transitions

3-Axis Accelerometer

Feature Extraction
y = y-axis Acceleration
z = z-axis Acceleration

Noise Reduction

Admission Control
threshold <= y <= threshold
threshold <= z <= threshold

Sirens

Microphone

Windowing

High Pass Filter

FFT

Feature Extraction
f = dominant frequency
y = magnitude of f
n = mean magnitude of FFT bins

Admission Control
threshold <= f <= threshold
m/n >= threshold

Music and Speech

Microphone

Windowing

Stat Function
Vamp = Variance of Amplitude

Admission Control
threshold <= Vzcr <= threshold
threshold <= Vamp <= threshold

Microphone

Windowing

Feature Extraction
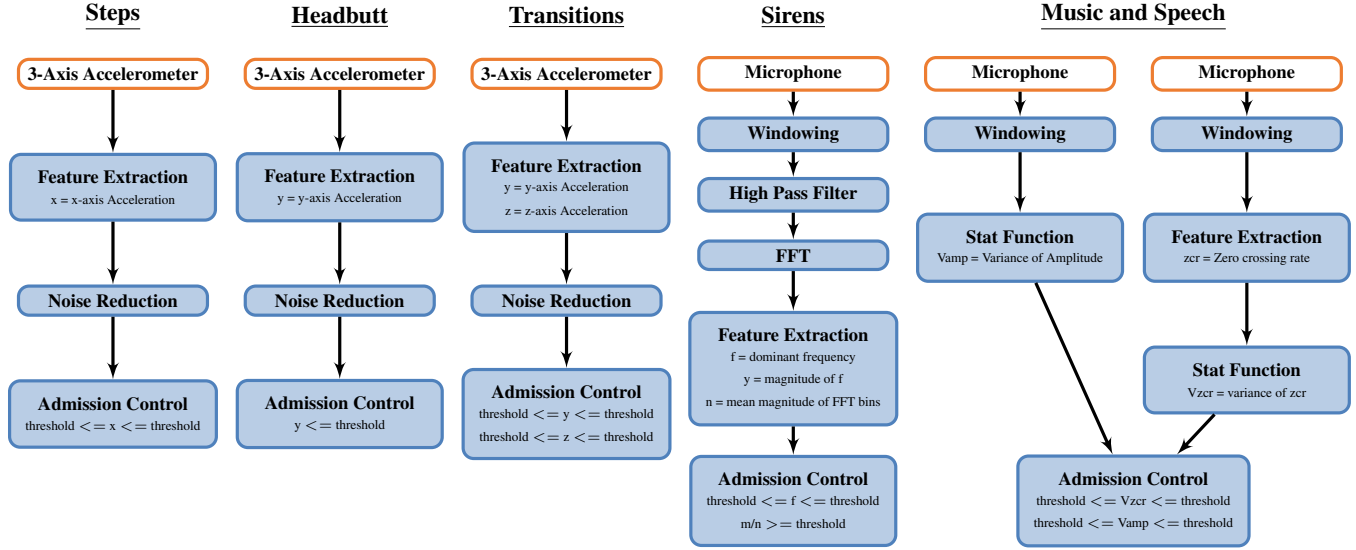zcr = Zero crossing rate

Stat Function
Vzcr = variance of zcr

Figure 3: Wake-up conditions pipelines for each of the applications.

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.

Headbutts Detects a sudden forward head movement. The application monitors the y-axis acceleration and searches for local minima between $-3.75\ m/s^2$ and $-6.75\ m/s^2$.

3.7.2 Audio Applications

We developed the following three microphone based applications.

Siren Detector Detects sirens originating from emergency vehicles. The application applies a 750 Hz high-pass filter in order to remove a significant potion of sounds that aren't sirens. The data in each window is transformed to the frequency domain using a FFT in order to extract the magnitude of the dominant frequency and the mean magnitude of all frequency bins. The ratio of the magnitude of the dominant frequency and the mean frequency is used to determine if the window contains pitched sounds. Pitched sounds between 850 Hz and 1800 Hz that last longer 650 ms are classified as sirens.

Music Journal Creates a list of all the songs heard during the day using the web services provided by Echoprint.me [4]. Audio data is partitioned into windows and passed to two branches for feature extraction. The first branch computes the variance of the amplitude over the entire window. The second branch further partitions the data into smaller windows and computes the zero crossing rate (the rate at which the signal changes from positive to negative or vice versa) for each sub-window. It then calculates the variance in zero crossing rate across the set the sub-windows. Finally, an admission control step uses thresholds (different for music and speech detection) on the extracted features to determine if an event of interest has occurred. Data is then passed to the Echoprint.me web service to identify the song.

Phrase Detection Similar to Music Journal, except different parameters are used in the wake-up condition and Google Speech API was used for speech-to-text translation.

We created a wake-up condition for each of these six applications using the processing algorithms described in section 3.6. Fig-

ure 3 shows the conceptual pipeline representation for each condition we constructed. Each one ends with an admission control step with configurable thresholds. The wake-up condition is satisfied when the relevant data or extracted features meet the admission control threshold.

3.8 Discussion

This section describes important questions that will need to be answered by hardware vendors to implement Sidewinder.

Identifying processing algorithms. Defining the appropriate set of common algorithms that should be included in the API and executed on the low-power processor for each sensor is a key challenge. First, there is a trade-off between algorithm generality and accuracy. Simple generic algorithms can support a large set of applications, albeit no specific application is likely to experience optimal performance. Conversely, a highly specialized algorithm may provide optimal performance but is only applicable to a limited set of applications. Second, there is also a trade-off between algorithm complexity and power savings. More complex algorithms can reduce energy consumption by preventing unnecessary wake-ups due to increased accuracy. On the other hand, more complex algorithms have higher computational demands, which require a larger and hungrier peripheral processor.

While determining the complete set of algorithms to be included as part of the runtime is beyond the scope of this paper, we anticipate that it will include algorithms for windowing, data filtering, feature extraction, admission control and transformations. Ideally, this set of algorithms should be standardized by the platform (ex. Android or iOS).

Access to sensor data. A related question is determining what data the sensor hub should pass to the application following a wake-up. Some applications may be interested in the raw sensor data, while others may want to use the filtered data or extracted features. Ideally, an API would allow developers to specify what data their application should receive when an event of interest occurs. Our current implementation passes a buffer of raw sensor data to the application.

Sizing. When creating the sensor node of the prototype implementation we evaluated two microcontrollers having different

power consumption levels. We noticed that the lower power microcontroller was not able to run some algorithms (such as Fast Fourier Transforms) in real-time. Determining the optimal number, type and size of processors to include in the sensor hub is an open research question. Each sensor (or small group of related sensors) may be supported by its own dedicated low-power processor. Alternatively, a larger processor could be used to serve the entire sensor hub. Identifying a sweet spot between the maximum number of concurrent algorithm executions, energy budget, cost and physical size of the sensor node is an open challenge and is a decision the hardware manufacturer will have to make.

Sensor fusion. Fusing inputs from multiple sensors is a common technique used for improving the accuracy of sensing applications. Whether low power sensor hubs should include support for sensor fusion, however, is not clear. On the one hand, such support could increase energy efficiency by reducing the occurrence of unnecessary wake-ups. On the other, sensor fusion tends to be application specific and the added complexity may negate any energy benefits. The current implementation allows sensor fusion at the main processor level. Once a wake-up condition triggers and passes raw data to the application, the application has the ability to run sensor fusion algorithms on the data.

4. Evaluation

Our evaluation is based on a trace-driven simulation. We measured power usage for our hardware to create a power model and collected accelerometer and audio traces. This data was fed into our simulator which modeled the behavior and power consumption of our devices under various configurations and applications.

We power profiled the Google Nexus 4 in order to create a model to estimate power consumption based on the outputs from the simulator. The results of the power profile are summarized in Table 1. During all the measurements, the devices' screen, WiFi and GPS were turned off. While the device is sleeping, its power usage is very low, consuming only 9.7 mW. While awake, the power consumption is significantly higher, averaging 323 mW. During our power measurements we noticed that additional energy is consumed during transitions between the asleep and awake states. Each transition takes about 1 second. During a wake-up transition, the average power consumption goes up to 384 mW, while during an awake-to-asleep transition the average power consumption is 341 mW.

We implemented the low-power processor on two different microcontrollers. One was a Texas Instruments (TI) MSP430 and the other a TI LM4F120. The MSP430 has the advantage of requiring very little power, consuming only 3.6 mW while awake. However, it has limited memory and cannot perform complex analysis of sensor data in real-time. In our tests, it was unable to run the FFT-based low-pass filter in real-time. The TI LM4F120 is powered by a Cortex-M4 processor. It can batch a higher number of accelerometer readings and can run all our filters in real time. However, this microcontroller has an energy footprint an order of magnitude greater than the MSP430, consuming an average of 49.4 mW while awake.

4.1 Trace Collection

Audio traces We collected three half-hour audio traces in different environments: an office, a coffee shop and outdoors. We used audio mixing software to add audio events of interest to the collected traces. The audio events of interest include music (5% of each trace), speech (5% of each trace), and sirens (2% of each trace). The events of interest were randomly selected from a library of audio files.

Human accelerometer traces We collected six hours of accelerometer traces from three different individuals while they perform routine daily activities: morning commute using public tran-

Figure 4: Aibo robotic dog used for data collection

sit, working in a retail store, and working in an office. Between 20% and 37% of each trace is spent walking.

Robotic accelerometer traces We collected synthetic traces by having a robot perform multiple runs with a prototype smartphone attached to its back. For each run, the robot logged the start and end of each action, which we use as the ground truth for our experiments. The smartphone ran an application that kept the device always awake and continuously recorded accelerometer readings for all three axes.

To enable us to conduct controlled and repeatable experiments, we mounted the prototype smartphone on the back of an AIBO ERA 210 robot dog (see Figure 4). Because the robot's actions can be scripted, this setup provides an efficient and reliable way to determine ground truth. In contrast, labeling data collected from human subjects with ground truth is error prone and labor intensive.

In each run, the robot performed five different actions: standing idle, walking, sit-to-stand transitions, stand-to-sit transitions, and headbutts. We created runs with three different levels of activity. Runs in groups 1, 2 and 3 spent 90%, 50% and 10% of the time standing idle, respectively. The reminder of the time was allocated as follows: 73% for walking, 24% for transitions between sitting and standing, and 3% for headbutts. This setup allows us to experiment with detecting actions that are common, somewhat frequent, and rare. In total, the robot executed 18 different runs: 9 for group 1, 6 for group 2 and 3 for group 3. We generated more runs for groups 1 and 2 because of the lower activity levels compared to group 3. To eliminate bias, the list of actions was generated randomly for each run, based on the expected probabilities of each action.

While our robotic testbed allows us to run live experiments, we chose instead to use trace-based simulation for several reasons. First, it took the robot close to an hour to complete a single experiment. Secondly, a thorough exploration of the configuration space of the various sensing approaches we consider would have required months of continuous live experiments. Moreover, taking fine grain power consumption measurements while the robot is in motion is not trivial.

4.2 Configurations

We used the simulator to evaluate the recall and precision of our applications under the following configurations.

- **Duty Cycling** The applications wake-up at fixed time intervals to collect sensor data for 4 seconds and run the event detection

State	Average Power Consumption (mW)	Average Duration
Awake, running sensor-driven application	323	N/A
Asleep	9.7	N/A
Asleep-to-Awake Transition	384	1 second
Awake-to-Asleep Transition	341	1 second

Table 1: Google Nexus 4 power profile.

algorithms. If an action is detected, the phone is kept awake for another 4 seconds, otherwise it goes to sleep for N seconds. N is referred to as the sleep interval. For our experiments, we use a sleep interval of 2, 5, 10, 20 and 30 seconds. As the sleep interval increases, more power is saved but recall suffers.

- **Batching** Similar to Duty Cycling, except when the phone is asleep sensor data is cached. When the device wakes, a batch of data from the sleep cycle is given to the application. We use the same sleep interval for Batching Cycling as we did for Duty Cycling.

- **Predefined Activity** This configuration simulates the Android's built-in significant motion detector. We constructed simple classifiers to wake up the device and invoke the callback method in the application when significant activity is detected (significant acceleration or sound).

- **Sidewinder based Classifier** For each of the applications, we constructed wake-up conditions to invoke the application when events of interests are detected.

- **Oracle** A hypothetical ideal implementation that only wakes up when the event of interest occurs. Such a wake-up condition would achieve perfect detection precision and recall, with the lowest possible power consumption. The difference between the power consumption of this method and the Sidewinder configuration provides an upper bound on the potential additional benefits of custom code offloading.

4.3 Metrics

For each sensing approach and trace, the simulator calculated the amount of sleep and awake time, the total number of wake-up events, and the recall and precision of the application. Using this data and the energy model derived from measurements of our prototype, we estimate the average power consumption. For the *Duty Cycling* experiments, the power model accounts only for the energy consumption of the Nexus 4. For *Batching* and *Predefined Activity*, the model also includes the cost of a low-power TI MSP430 microcontroller. Finally, experiments configured to use *Sidewinder* include the cost of the TI MSP430, with the exception being the siren detector which required the more powerful TI LM4F120 to run FFT in real time.

5. Results

In this section we present the results of simulations conducted on the accelerometer and audio traces described in Section 4.1. We answer the following questions:

1. How much power can be saved with more energy efficient sensing approaches?

2. How close to optimal is Sidewinder?

3. How does Sidewinder compare to Predefined Activities?

4. How well do Duty Cycling and Batching perform?

Wake-up Mechanism	Sirens	Music	Phrase
Oracle	16.8	27.2	14.7
Predefined Activity	51.9	51.9	51.9
Sidewinder	63.1*	32.3	35.6

* Includes the more powerful TI LM4F120

Table 2: Average power consumption for the audio applications. Values measured in milliwatts.

5. How representative are the accelerometer experiments on the AIBO of expected performance with humans?

Figure 5 presents the power usage, relative to Oracle, of replaying the synthetic accelerometer traces under the various sensing configurations. For each configuration[1], the graph presents power consumption over Oracle. Results are averages across runs of the same group. In order to make it easier to compare across approaches, we calibrated all approaches so that they all achieve 100% recall. Duty Cycling is the one approach that cannot achieve 100% recall with any reasonable sleep interval. Figure 6 shows the recall for Duty Cycling at 90% idle. All sensing approaches achieved similar average precision (Headbutts: 89%, Transitions: 91%, Walking: 93%).

Table 2 shows the average results from running the the simulations on the collected audio traces. We omitted the results for Duty Cycling and Batching because they are similar to the results from the simulations on accelerometer traces.

5.1 How Much Power can be Saved?

The Oracle in our work is a hypothetical ideal which is in a sleep state most of the time and only wakes up when events of interest occur. It has perfect recall, precision and the best possible power usage. Always Awake, on the other hand, never sleeps and therefore, and has the worst possible power usage. The difference in power usage between these two approaches represents the power that could be saved by better sensing approaches. Always Awake consumed on average 323mW of power. In our most demanding scenario, step detection with 10% idle, the Oracle consumed on average 266mW. For the least demanding, step detection with 10% idle, the Oracle consumed on average 16.4mW. This means, depending on the scenario, *there is potential to reduce power consumption by 17.7% to 94.9%*.

5.2 How Close to Optimal is Sidewinder?

By comparing the performance of the Sidewinder approach to Oracle we observe that the Sidewinder approach achieves between 92.7% and 95.7% of the of the possible power savings [2] for our

---

[1] Results for Batching are shown with a 10s sleep interval as the other results were similar to Duty Cycling

[2] $(AlwaysAwake - Sidewinder)/(AlwaysAwake - Oracle)$
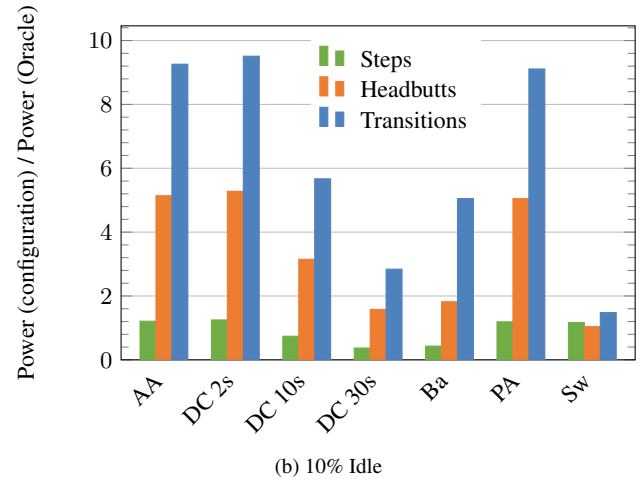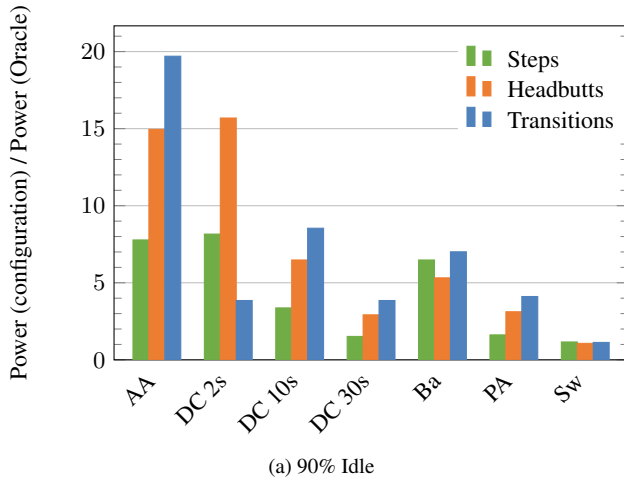
(a) 90% Idle

(b) 10% Idle

Figure 5: Power usage of configurations: Always Awake (AA), Duty Cycling (DC) with various sleep intervals, Batching (Ba) with 10s sleep interval, Predefined Activity (PA) and Sidewinder (Sw) relative to Oracle for synthetic accelerometer traces
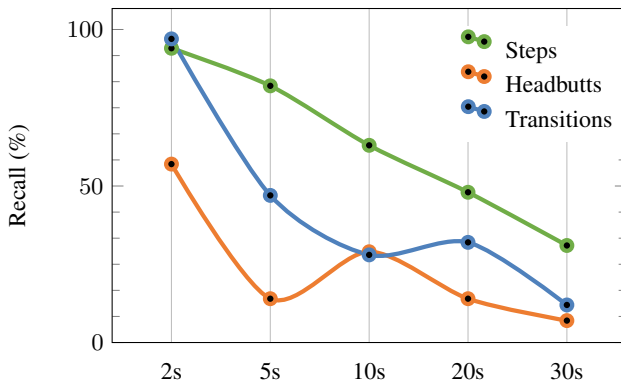

Figure 6: Recall for Duty Cycling on synthetic accelerometer traces with 90% idle

accelerometer-based applications. Audio applications performed similarly with saving between 85% and 98%.

The suboptimal nature of wake-up conditions is illustrated by the phrase detection application. Whereas, the Oracle only wakes up when the phrase of interest occurs (<1% of each trace), our wake-up condition powers up the device every time it detects a speech segment (approximately 5% of each trace). However, even with this limitation, Sidewinder achieves 93% of the possible energy saving for this application.

We conclude *it is possible to build a wide range of classifiers based on a set of generic processing algorithms, and that the resulting classifier achieves the large majority of available power savings. Moreover any additional power saving that custom code may achieve are likely to be very limited*.

5.3 Sidewinder vs. Predefined Activity

To make the comparison to Predefined Activity as fair as possible, we explored the parameter space to determine the best thresholds for *significant acceleration* and *sound intensity*[3]. We chose values that minimize power consumption, while maintaining 100% detection recall. Thus the parameters used in this scenario are over-fitted

to our test data and represent a best case scenario that skews the results in favor of Predefined Activity.

As expected, the power consumption resulting from the use of significant activity detectors (significant sound, significant motion) are proportional to the amount of activity in the trace and the popularity of the event of interest.

In the accelerometer experiments, Predefined Activity has similar power consumption to Sidewinder for *steps*, which is a common event, but consumes 4.7 and 6.1 times more power to detect *headbuttss* and *transitions*, which are less frequent events. In the experiments performed on the audio traces, Predefined Activity consumed 18% less power for *sirens* than Sidewinder, but 45% and 60% more power for *music journal* and *phrase detection*, respectively. Due to the higher complexity of the wake-up condition used for siren detection, the power consumption model had to account for the powerful TI LM4F120 microcontroller instead of the MSP430, which consumes an additional 40 mW.

We conclude that a *a small number of predefined activities are unlikely to support efficiently a wide range of applications. This is particularly the case for applications interested in infrequent events*.

5.4 Sidewinder vs. Duty Cycling and Batching

Duty Cycling performs poorly. Short sleep intervals actually result in an increase in power consumption (339 mW compared to an average of 323 mW for Always Awake) due to frequent transitioning between awake and asleep states. Longer sleep intervals are more effective at saving energy, but they do so by sacrificing recall. For example, a sleep interval of 10 seconds reduces the Headbuts and Transitions recall bellow 30%.

Batching achieves perfect recall, but requires long batching intervals to achieve large energy savings. Therefore, this approach is not appropriate for applications with timeliness constrains. For example, the user of a gesture recognition application [20, 29] would not be satisfied if the application detects the performed gesture after a delay of more than a couple of seconds. We anticipate that in practice realistic batching intervals are in the order of a few seconds, depending on the sensor data acquisition rate and the size of

---

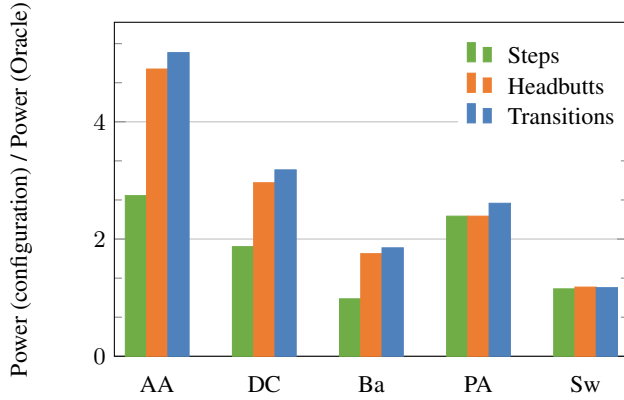[3] Two predefined activities supported by our hardware

Figure 7: Power usage of configurations: Always Awake (AA), Duty Cycling (DC), Batching (Ba), Predefined Activity (PA) and Sidewinder (Sw) relative to Oracle for human traces

the data buffer. Additionally, the device often wakes up to find out that no events occurred in the current batch.

In most cases *Duty Cycling and Batching consumed 2.4 to 7.5 times more power than Sidewinder. To achieve significant power saving, Duty Cycling and Batching have to either sacrifice recall or timeliness.*

5.5 Human Traces

Figure 7 shows the results from running the step detector application on traces collected from three human subjects. Since these traces are not annotated with ground truth, we use the steps detected by an *Always Awake* configuration as the baseline for determining recall. For Duty Cycling and Batching we show only a sleep interval of 10 seconds. All approaches except Duty Cycling (82%) had 100% recall.

The results from these experiments show benefits very similar to the synthetic experiments for runs with low and medium levels of activity. The Sidewinder approach achieves at least 91% of the available power saving in each of the traces.

Additionally, we note that the generic wake-up condition performs poorly. We attribute the relatively high power consumption to the fact the human subjects were performing a wide range of activities. While most of the activities were not events of interest, they resulted in unnecessary wake-ups.

6. Related Work

The idea of waking up a device when an event of interest occurs has been around since the inception of mobile phones. The phone's radio transceiver wakes up the device when an incoming call or a text message is received [5]. Wake on Wireless [30] extended this idea by augmenting a PDA with a low-power radio that would send a wake-up message when an incoming call is received. Similarly, Wake on WLAN [24] allows remote wake-up of wireless networking equipment.

Turducken [31] generalizes the "wake on event of interest" approach to several types of applications and to multiple components operating at increasingly small power-levels. Little Rock [27] applies Turducken's multi-tiered architecture to sensing on mobile devices. Reflex [19] complements the idea proposed by Turducken by providing a shared memory abstraction to be used by the different processors. Little Rock and Reflex expose application developers to the heterogeneous architecture. In contrast, Sidewinder hides the heterogeneous nature of the system from the application developer.

Creating an application that makes use of Sidewinder does not require the developer to write low-level code for the low-power sensor hub. Instead, developers create custom wake-up conditions by configuring pipelines of commonly used algorithms. This approach increase portability, while achieving the majority of the potential power savings.

Smartphone manufacturers have started to incorporate low-power processors into their architectures, but have only implemented limited APIs that provide fixed functionality. Apple's M7 and M8 motion co-processors are used to collect, process, and store sensor data even while the main CPU is asleep and applications can retrieve historical motion data via the CoreMotion API [3]. Some recent Android devices allow batching of sensor readings [1], and the Motorola Moto X provides recognition for a small number of predefined activities that can be used as wake-up conditions [7, 10]. While these wake-up conditions work well for some applications, they are inefficient for many other types of applications that are not interested in the set of predefined activities. In contrast, Sidewinder supports a wide variety of applications by providing developers an easy mechanism to create custom wake-up conditions.

Most of the previously noted works focused on system architecture modification in order to lower the cost of sensing. Alternative approaches have also been explored. Ace [25] is a middleware that supports continuous context-aware applications while mitigating sensing cost for acquisition of context attributes (such as AtHome and IsDriving). It achieves power savings when multiple applications request strongly correlated context attributes. Additionally, it can reduce power consumption when a "cheaper" sensor exists, which can determine the value of a different context attribute that has a strong correlation with the requested context attribute (e.g. use the accelerometer to check if the user is jogging instead of using the GPS to determine if the user is at work). A middleware such as Ace is a great example of a library that can run on top of Sidewinder and achieve additional power savings. Sensor fusion has also been an active focus of related research. Data from multiple sensors can be used to increase context-awareness in mobile devices [12, 14].

While our focus was on power-efficient acquisition of sensor data, next generation mobile perception applications face related problems regarding partitioning of application code. MAUI [13] enables fine-grained energy-aware offload of mobile application code to remote servers. Similarly, Odessa [28] uses code-offloading to address the issue of processing sensor data on resource constrained mobile devices.

7. Conclusion

In this paper we proposed *Sidewinder*, a new approach for continuous mobile sensing. In this approach, the platform implements common sensor data processing algorithms that execute on a low-power processor, and application developers construct wake-up conditions for events of interest by selecting among the set of predefined common processing algorithms and tuning their parameters. We presented an extensive evaluation showing the benefits of using Sidewinder as wake-up mechanisms for the multiple accelerometer and audio-based applications.

Our immediate future work includes developing an FPGA-based prototype, performing a thorough exploration of what algorithms should be included as part of the platform and analyzing their power and computational requirements. We would also like to explore supporting multiple concurrent applications while still maintaining predictable performance. When receiving multiple wake-up conditions, the sensor manager can attempt to improve performance by combining the pipelines that use common algorithms.

Another interesting extension includes adding "smartness" to the low-power sensor hub. Application developers may face challenges in selecting the optimal algorithms and configuration parameters for their wake-up conditions. But given feedback from the more complex algorithms running on the application level, self-learning mechanisms may be able to tune the paramers used on the wake-up conditions. It is easy to imagine an application notifying the sensor hub about wake-ups when events of interest were not actually detected (i.e. false positives). However, it will be more difficult to automatically identify events of interest missed by the wake-up condition running on the low-power node (i.e. false negatives).

References

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

[2] Android motion sensors. `http://developer.android.com/guide/topics/sensors/sensors_motion.html`, .

[3] Core motion framework reference. `https://developer.apple.com/library/ios/documentation/coremotion/reference/coremotion_reference/index.html`.

[4] Echoprint - open source music identification. `http://echoprint.me/`.

[5] Qualcomm - 3g/4g connectivity (gobi). `https://developer.qualcomm.com/mobile-development/maximize-hardware/3g4g-connectivity-gobi`.

[6] I2c. URL `https://en.wikipedia.org/w/index.php?title=I%C2%B2C&oldid=670659499`. Page Version ID: 670659499.

[7] Moto x. `http://www.motorola.com/motox`.

[8] SensorManager | android developers. URL `https://developer.android.com/reference/android/hardware/SensorManager.html`.

[9] Universal asynchronous receiver/transmitter. URL `https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&oldid=673411359`. Page Version ID: 673411359.

[10] X8 mobile computing system. `http://www.motorola.com/us/X8-Mobile-Computing-System/x8-mobile-computing-system.html`.

[11] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[12] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 361–365. IEEE, 2004.

[13] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[14] H. W. Gellersen, A. Schmidt, and M. Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7(5):341–351, 2002.

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

[16] J.-y. Hong, E.-h. Suh, and S.-J. Kim. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509–8522, 2009.

[17] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 125–138. ACM, 2006.

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

[19] X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *Proc. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2012.

[20] J. Liu, L. Zhong, J. Wickramasuriya, and V. Vasudevan. uwave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing*, 5(6):657–675, 2009.

[21] N. Maisonneuve, M. Stevens, M. E. Niessen, P. Hanappe, and L. Steels. Citizen noise pollution monitoring. In *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government*, pages 96–103. Digital Government Society of North America, 2009.

[22] N. Maisonneuve, M. Stevens, M. E. Niessen, and L. Steels. Noisetube: Measuring and mapping noise pollution with mobile phones. In *Information Technologies in Environmental Engineering*, pages 215–228. Springer, 2009.

[23] S. E. Minson, B. A. Brooks, C. L. Glennie, J. R. Murray, J. O. Langbein, S. E. Owen, T. H. Heaton, R. A. Iannucci, and D. L. Hauser. Crowdsourced earthquake early warning. 1(3): e1500036–e1500036. ISSN 2375-2548. doi: 10.1126/sciadv.1500036. URL `http://advances.sciencemag.org/cgi/doi/10.1126/sciadv.1500036`.

[24] N. Mishra, K. Chebrolu, B. Raman, and A. Pathak. Wake-on-wlan. In *Proceedings of the 15th international conference on World Wide Web*, pages 761–769. ACM, 2006.

[25] S. Nath. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Proc. of the 10th Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 29–42. ACM, 2012.

[26] D. Preuveneers and Y. Berbers. Mobile phones assisting with health self-care: a diabetes case study. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 177–186. ACM, 2008.

[27] B. Priyantha, D. Lymberopoulos, and J. Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE*, 10(2):12–15, 2011.

[28] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[29] T. Schlömer, B. Poppinga, N. Henze, and S. Boll. Gesture recognition with a wii controller. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages 11–14. ACM, 2008.

[30] E. Shih, P. Bahl, and M. J. Sinclair. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proc. of the 8th Conference on Mobile Computing and Networking(MobiCom)*, pages 160–171. ACM, 2002.

[31] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proc. of the 3rd Conference on Mobile Systems, Applications, and Services (MobiSyS)*, Seattle, WA, June 2005.

[32] C. C. Tsai, G. Lee, F. Raab, G. J. Norman, T. Sohn, W. G. Griswold, and K. Patrick. Usability and feasibility of pmeb: a mobile phone application for monitoring real time caloric balance. *Mobile networks and applications*, 12(2-3):173–184, 2007.