The case for simple configurable classifiers for efficient continuous sensing on mobile devices

Silviu Jingoi
University of Toronto

Eyal de Lara
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Applications that perform continuous sensing on mobile phones have the potential to revolutionize everyday life. Examples range from medical and health monitoring applications, such as pedometers and fall detectors, to participatory sensing applications, such as pollution and traffic monitoring. Unfortunately, current mobile devices are a poor match for continuous sensing applications as they require the device to remain awake for extended periods of time, causing fast battery drainage. This paper proposes a new approach that offloads a simple configurable event detection classifier to a low power processor and wakes up the main processor only when an event of interest occurs. This approach differs from other heterogeneous architectures in that developers are presented with a programming interface that lets them construct application specific classifiers by from a set of predefined pre-processing algorithms and tuning their parameters.

Q: Add conjecture? add summary of validation to support conjecture?

1. INTRODUCTION

Today, smartphones and tablets are used primarily to run interactive foreground applications, such as games and web browsers. As a result, current mobile devices are optimized for a use case where applications are used intermittently during the day, in sessions that last for several minutes. For example, during her break Alice picks up her phone, checks the weather, reads the latest news and plays a game for a few minutes, and then puts it away to go back to work. To maximize battery life, current mobile devices are designed to go into sleep state for most of the day when they are not supporting such interactive usage.

Unfortunately, most mobile platforms are a poor match for a growing class of mobile applications that perform continuous background sensing. Examples range from context-aware applications [5, 7], such as medical applications [6, 17, 20] that improve our well-being or even save lives using activity recognition (e.g., fall detection), to applications that use participatory sensing to get a better understanding of the physical world, such as noise pollution monitoring [13, 14] or traffic prediction [8]. While the processing demands of these applications are modest most of the time, they require periodic collection of sensor readings, which prevents the device from going to sleep for extended periods of time. As a result, applications that perform continuous sensing may cause the device's battery to drain within several hours.

To improve support for continuous sensing applications the research community has proposed the use of fully programmable heterogeneous architectures [12, 18, 19]. In these approaches, developers partition their applications to offload the initial 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 an event of interest, it proceeds to wake up the phone and passes control to the rest of the application. While the ability to run custom code on the low-power hardware provides great flexibility, the significant complexity inherent in this approach has so far prevented its adoption in commercial devices. Instead, smartphone manufacturers, realizing the potential of sensing applications, have recently incorporated low-power processors into their architectures, but have limited application developers to APIs that provide fixed functionality, by either batching sensor readings or recognizing a small number of predefined activities that can be used as wake-up conditions [2, 3, 4].

In this paper we argue that these APIs are insufficient for supporting a rich and flexible set of continuous sensing applications. Batching is inefficient for applications that depend on infrequent events and is not appropriate for applications that require crisp response time. An activity recognition API provides little flexibility with no support for applications interested in events that are not covered by the set of predefined activities.

Instead, we argue that the sensing API should expose access to pre-processing algorithms on a low-power processor. Data pre-processing includes data cleaning, normalization, transformation, feature extraction and selection, etc [10] Q: Previous sentence was copied word for word from the paper's abstract. Is that ok? Need to format differently?. We propose an approach that enables programmers to create custom wake-up conditions by choosing among a set of predefined pre-processing algorithms, implemented on the low-power processor, and tuning their parameters. By providing developers with access to commonly used pre-processing algorithms (e.g., noise reduction, feature extraction), as opposed to higher-level activity detectors, this approach provide a better balance between flexibility and ease of deployment. This access allows developers to create a broad set of simple event classifiers that can be used to detect a wide range of activities. Simultaneously, they are significantly easier to program compared to fully programmable offloading. Moreover, since the pre-processing algorithms are pre-specified, their implementations can be optimized (by the device manufacturer) for each low-power processor, improving application portability between devices. Furthermore, it is possible to combine the pre-processing and wake-up operations when multiple applications register interest in the same set of events. While simple classifiers could be programmed directly by individual applications, it is easy to imagine activity recognition libraries that take advantage of the functionality of our approach, to provide detection algorithms for a large number of activities.

The rest of this paper is organized as follows. Section 2 discusses existing approaches to continuous sensing and their limitations. Section 3 proposes a conjecture, describes our new approach for continuous sensing and identifies some of its advantages and challenges. Section 4 highlights the limitations of existing sensing approaches and provides some evidence that supports the proposed conjecture. Finally, Section 5 conclude the paper.

2. BACKGROUND

Mobile application developers can choose among several existing approaches when attempting to perform continuous sensing. The availability of some of these approaches depends on the mobile device's hardware and operating system.

2.1 Always Awake

The basic approach for continuous sensing is to keep the mobile device *always on*. As an example, on the Android operating system, the sensing application acquires a wake lock to prevent the device from sleeping and registers a listener for a specific sensor. Whenever the sensor produces a new reading, the operating system passes the reading to the application via a callback function. This operation continues until the listener is unregistered or until the wake lock is released. The benefit of this approach is that it achieves the highest possible accuracy because all the sensor data is delivered to the application. However, the device cannot be placed in a sleeping state and hence, the power consumption of this approach is high.

2.2 Duty Cycling

To avoid keeping the device awake for extended periods of time, *duty cycling* allows a device to sleep for fixed, usually regular, periods of time. An application on Android can implement duty cycling by scheduling a wake-up alarm using the system's alarm service and by ensuring that it has not acquired a wake lock. When no application has a wake lock, the operating system can power down the processor. When the wake-up alarm fires, the processor wakes up and notifies the application. Next, the application collects sensor readings for a short period of time, and then reschedules another alarm.

Duty cycling has several drawbacks. While the device is sleeping, events of interest will not be detected because the application does not receive any sensor data that were produced at those times. Additionally, during the short time periods when the device is awake and collecting sensor readings, the events of interest might not occur. These power transitions are wasteful and expensive.

The basic duty cycling approach can be improved by dynamically adjusting the wake-up time. When an application is woken up and detects an event of interest, it continues to collect sensor readings until events do not occur for some period, after which it returns to its regular duty cycling schedule. This improves detection accuracy (e.g., event recall) but at the expense of increased energy consumption. Unfortunately, finding a good balance between accuracy and power savings depends on the specific application and the user's behaviour. Another drawback of this approach is that it does not scale well with multiple applications. It is possible for mutually unaware applications to implement conflicting wake-up schedules, resulting in little or no sleep.

2.3 Sensor Data Batching

An extension to duty cycling is *sensor data batching* in which the device hardware is able to collect sensor readings while the main processor is asleep. Upon wake-up, the entire batch of sensor data is delivered to the application(s) that registered a listener for that specific sensor. Unlike duty cycling, batching requires hardware support (e.g., it is currently only available on the Google Nexus 5, running Android 4.4 and the iPhone 5S, 6 and 6S), but it enables applications to receive all the sensor data, and hence provides detection accuracy similar to the Always Awake approach. However, batching affects detection timeliness. Applications may detect events

Approach		Advantages	Disadvantages
Always Awake		Highest Recall	Highest power consumption
Duty Cycling		Significant power savings	Significant loss of event of interest recall Power inefficient for infrequent events of interest
Batching		Significant power savings Recall matching *Always Awake*	Significant delay between occurrence and detection of event Power inefficient for infrequent events of interest
Computation Offloading		Significant power savings	Programming complexity Security concerns Difficult to support multiple applications
Predefined Event Wake-ups	High-level activity	Easy to use High recall Significant power savings	Only usable by a small set of applications
	Significant Motion	Easy to use High recall	High proportion of unnecessary wake-ups when event of interest is infrequent

Table 1: Summary of advantages and disadvantages for various continuous sensing approaches

of interest many seconds after they actually occurred. Also, as with the duty cycling approach, the device may wake up to find out that no events of interest have occurred in the current batch, which is wasteful.

2.4 Computation Offloading

Generalizing batching, a low-power peripheral processor can be used to collect and *compute* on sensor data while the main processor is asleep [12, 19]. This approach allows developers to offload their own sensor data analysis algorithms to the peripheral processor, enabling arbitrarily rich sensing applications. However, developers are exposed to the full complexity of the architecture, making application development more difficult. Even simple sensor-driven applications have to be refactored into distributed programs, and the code that is offloaded has to be chosen carefully so that it can run in real time. The latter is complicated because it depends on the type and functionality of the peripheral processor that is available. As a result, multiple versions of the application may be required to handle hardware variations across phones. Furthermore, this approach may create security concerns because applications are allowed to run arbitrary code on the peripheral processor. Additionally, since each application is written independently, this approach makes it hard for multiple applications to program or use the same sensor.

2.5 Activity Detection

Given the challenges with full computation offloading, recent mobile devices limit offloading to a small, predefined set of *activity detection* algorithms. For example, the Motorola Moto X [4] uses two low-power peripheral processors to wake up the device when certain events occur. A dedicated natural language processor is used to wake up the device when a user says the phrase "OK Google Now". Also, a contextual processor is used to turn on a part of the screen when the mobile device is taken out of a pocket, or start the camera application when the processor detects two wrist twists. Additionally, Android 4.4 [1] and Apple's CoreMotion [3] framework supports specific wake-up events based on detecting "significant motion" or steps, if the device has the appropriate sensors. However, the application developer has no control over the underlying activity detection algorithms, including customization of parameters. This approach is easy to use, and it provides good energy savings because the device only wakes up when the event of interest occurs. However, it only works for applications that can take advantage of the predefined set of activities, limiting the number and types of applications that can be supported.

3. UNNAMED SECTION

3.1 Observation

Continuous sensing applications are commonly structured as modular pipelines of pre-processing steps, followed by a classifier. Moreover, while the application logic of different sensing applications varies widely, applications that use the same sensors tend to make use of similar algorithms in the early stages of their event detection pipelines. However, different detection applications or different calibrations of the same application may run these algorithms with different parameter configurations. For example, acceleration data has significant noise, and hence many applications have implemented low-pass filters to smoothen the accelerometer readings. In addition, audio data is commonly passed though noise reduction steps to remove background noise or through high-cut or low-cut filters in order to reduce the amount of treble or bass in the sound. Image processing makes use of low-pass, high-pass and band-pass filtering for noise reduction, and edge detection [16, 15].

Feature extraction is another common step for event detection algorithms. The features extracted from the sensor data are commonly used for admission control or by the event classifier. While there exists some overlap, the set of extracted features generally varies between different sensing applications.

3.2 Conjecture

Continuous mobile sensing applications should be structured as a pair of classifiers of increasing complexity: A simple high recall and moderate precision classifier that runs continuously on low-power hardware, and acts as an energy efficient wakeup mechanism for a higher complexity classifier that provides both high recall and high precision, running on the main CPU of the device.

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

3.3 Approach

Our approach makes continuous mobile sensing possible by providing a sensor API that gives access to configurable pre-processing algorithms running on a low-power sensor node. Applications construct *simple classifiers* for events of interest by using a pre-defined set of common pre-processing algorithms and tuning their parameters. The *simple classifiers* would execute on the low-power sensor node and, when events of interest are detected, 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.

3.4 Advantages

This approach has multiple advantages over the approaches described in Section 2. Programming complexity is decreased because application developers can use the pre-defined pre-processing algorithms, rather than implementing them themselves. It also creates predictable performance for the algorithms executed on the low-power processor. Profiling each algorithm will provide metrics about computational and time requirements, as well as power consumption. Predictable performance is important in order to support multiple concurrent classifier executions or applications that make use of multiple sensors.

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 peripheral processor. Moreover, it ensures application portability to other devices featuring this sensing approach. Since the algorithms are pre-specified, the device manufacturer can optimize their implementations for each low-power processor, improving application portability between devices. Furthermore, it is possible to optimize the operation of multiple applications that use the same algorithms. Eyal: the same applies to the actual sensor as different instances of even the same model a device may use sensors from different manufacturers.

3.5 Challenges

The main challenge with our approach is defining the appropriate set of algorithms that should be included in the API and executed on the low-power processor for each sensor. First, there is a trade-off between algorithm generality and accuracy. Simple generic pre-processing 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. A related challenge 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. An across-the-board solution would be to allow the applications to specify what data they are interested in via the API.

We believe application developers may face challenges in selecting the optimal algorithms and configuration parameters for their *simple classifiers*. Given feedback from the application, self-learning mechanisms may be able to determine optimal configuration parameters for the algorithms used. Ashvin: training could be used 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 classifier running on the low-power node (i.e. false negatives).

Another challenge lies in supporting multiple concurrent applications while still maintaining predictable performance. The fully programmable offloading approach has a similar issue because multiple applications are not able to execute code concurrently on the peripheral processor. However, given that the algorithms in our approach are pre-defined, they can be profiled in terms of computational power requirements. A resource manager would be required to orchestrate and synchronize concurrent requests from multiple applications. This issue also has an impact on the number and sizing of processors contained by the sensor hub. Each sensor (or small group of related sensors) may be supported by its own dedicated low-power processor. Alternatively, a slightly bigger processor could be used to serve the entire sensor hub. Identifying a sweet spot between the maximum number of concurrent algorithm executions,

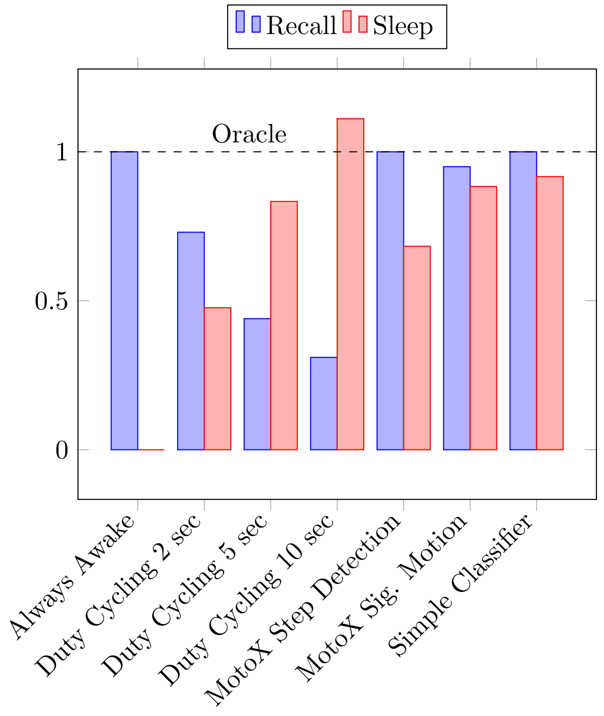

Figure 1: Footstep Detection

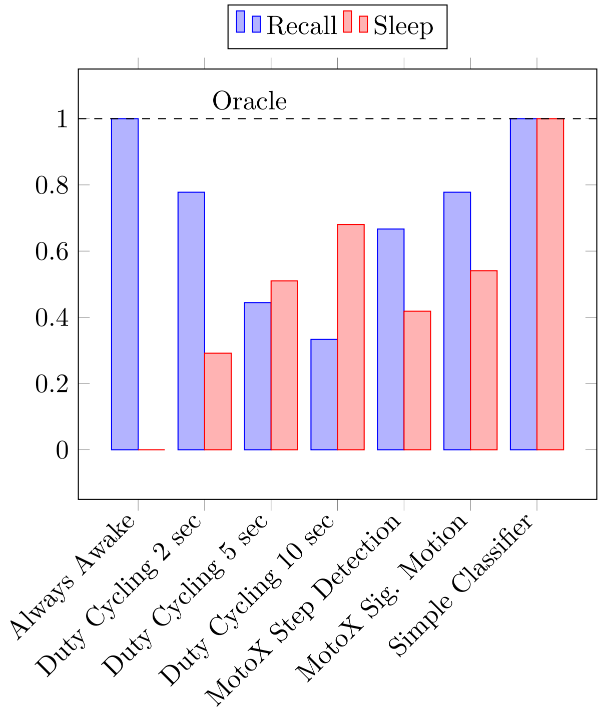

Figure 2: Fall Detection

energy budget, cost and physical size of the sensor node is an avenue for future work.

TODO: describe challenge in supporting applications interested in the multiple sensors (sensor fusion)

4. VALIDATION

Showing what algorithms should be implemented by the platform and defining the API will be part of our future work. In this section we provide evidence to support our conjecture. We implemented simple classifiers built using common pre-processing algorithms for two accelerometer-based applications. We evaluated the various sensing approaches using trace-based simulations. We present recall and sleep time metrics obtained from these simulations.

This section:

1. Highlights the limitations of the Duty Cycling and Batching approaches
2. Illustrates the inflexibility of Predefined Event detection featured by current generation smartphones
3. Demonstrates that simple classifiers built from common pre-processing algorithms can achieve close to ideal recall and device sleep time

4.1 Trace Collection

We collected two traces from human subjects using a Motorola Moto X device, running Android 4.4.4. The smartphone ran an application that kept the device always awake and continuously recorded all accelerometer readings and predefined event (significant motion and steps).

The first trace is about 10 minutes long and contains a combination of frequent events of interest (steps; about 40% of the trace) and infrequent events of interests (falls; about 2% of the trace). We annotated this trace with ground truth information about the time stamps when events of interest occurred.

The second trace is 2 hours and 50 minutes long representing a person's morning routine (getting ready, having breakfast and commuting to work). This trace was not annotated with ground truth information.

4.2 Applications

We developed two applications that detect activities performed by human subjects:

1. A *Step Detector* based on the human step detection algorithm proposed by Ryan Libby in [11].
2. A *Fall Detector* based on a simple thresholding algorithm, as described in [9].

4.3 Simple classifiers

We created *simple classifiers* for our step and fall detection applications.

The step detection classifier consisted of:

1. A *noise-reduction* step, based on exponential moving average.
2. A *feature extraction* step, computing the magnitude of the noise-reduced acceleration vector.
3. An *admission control* step, checking if the total acceleration magnitude exceeds a threshold.

The fall detector classifier was similar, with the following differences:

1. The *noise-reduction* step was milder, resulting in more detailed accelerometer data.
2. There were two *admission control* steps. It checks for acceleration magnitude below a small threshold (representing free-fall), followed by a spike in acceleration magnitude above a different threshold (representing the moment of impact).

4.4 Oracle

A hypothetical ideal implementation that only wakes up when an event of interest occurs. Such a wake-up condition would achieve perfect precision[1] and recall[2], with the highest possible sleep time. The difference between the power consumption of this method and our approach provides an upper bound on the potential additional benefits of custom code offloading.

This setup provides a baseline for comparison. Figures 1 and 2 show the event of interest recall and achieved sleep time for various sensing approaches, normalized by the recall and sleep time of the *Oracle*. We assumed that the Oracle achieves 100% recall for both activities and sleeps for 60% and 98% of the time for step detection and fall detection, respectively.

4.5 Always Awake

The applications keep the phone awake all the time, constantly collecting accelerometer readings. Under this sensing approach, both the step and fall detector applications achieved very high levels of precision and recall for the 10 minute trace. The recall and precision achieved by the *Step Detector* were 100% and 98%, respectively. The *Fall Detector* had perfect precision and recall.

4.6 Duty Cycling

We modified the applications so that they check sensor readings periodically and then put the phone to sleep. On wake-up, the phone is kept awake for 5 seconds in order to collect sensor data. This software-only implementation can run on any mobile device and does not require special hardware support. As expected, Duty Cycling is effective at reducing the amount of time the device is awake (and thus, power consumption), but performs poorly in terms of event of interest recall.

4.7 Batching

This configuration emulates hardware support available on the Nexus 5 and recent iPhone devices for collecting and batching accelerometer readings while the main processor sleeps. The application wakes up periodically, reads the batch of sensor readings, runs the detection algorithm(s), and goes back to sleep. Because the applications get access to all the sensor data, their event of interest recall would be the same as the Always Awake approach. However, the power efficiency of this approach is dependent on several device-specific characteristics. The size of the sensor data buffer creates an upper bound on the amount of time the device can be asleep, while ensuring that the application will eventually receive all the sensor data. The device's computational capabilities will have an impact on the amount of time it takes to process the data in a batch, and thus, the amount of time the device needs to be awake.

Batching is not appropriate for applications with timeliness constraints. For example, the user of a gesture recognition application [?, ?] would not be satisfied if the application detects the performed gesture after a delay of more than a couple of seconds. Additionally, the device would often wake up to find that no events have occurred in the current batch. We therefore conclude that batching will result in significant energy waste for applications interested in low frequency events (e.g., gesture recognition, fall detection).

4.8 Predefined Event Detection

This configuration makes use of the features available in Android 4.4+ and iOS 7.0+ for recognizing a limited number of predefined activities (significant motion and steps).

We compared the ground truth annotations with the step events detected by the Android framework. The step detector achieved perfect recall, however, its precision was approximately 70%. Upon inspection, we noticed that falls and other events such as sitting, standing and taking the device out of the pocket were incorrectly classified as steps. The lack of configuration options available in Android's sensor manager makes it impossible to calibrate the step detector for usage by specific users. As such, an application that requires high degrees of precision and recall would need to implement its own step detection algorithm.

We wanted to see if the significant motion events de-

[1]
$$Precision = \frac{TruePositives}{(TruePositives + FalsePositives)}$$

[2]
$$Recall = \frac{TruePositives}{(TruePositives + FalseNegatives)}$$

tected by the device could be used as wake-up conditions for either step or fall detection, effectively removing the need for simple classifiers running on a low-power processor. From a recall point of view, significant motion events were detected for nearly all the events of interest. However, these events were delivered to the application up to 10 seconds after the event occurred. Without access to past sensor data, the application is unable to determine if the activity that caused the significant motion event is indeed an event of interest. Unless combined with sensor data batching, these significant motion events would be useless for the application.

To determine the feasibility of the significant motion event as a wake-up condition for events of interest that occur infrequently, we analysed the 2 hour and 50 minute long trace. We assumed the event of interest (e.g., falls) did not occur during the entire length of the trace. For this trace, the significant motion event fired, on average, every 2.47 seconds. Using the significant motion event as a wake-up condition for infrequent events of interest would be very inefficient from a power consumption perspective. The device would wake up too frequently, without actually detecting any events of interest.

4.9 Simple Classifiers

We defined simple classifiers to act as custom wake-up conditions as described in Section 4.3. On wake-up, the applications acquire the raw sensor readings used by the classifier to detect the event and run their own detection algorithm, and go back to sleep.

Figures 1 and 2 show that this approach achieved perfect recall of both events of interest. The difference in sleep time between this approach and *Oracle* is attributed to unnecessary wake-ups resulting form false positives generated by the simple classifier. These false negatives were discarded by the detection algorithms implemented in the applications.

4.10 Limitations

Establishing a greater support for the proposed conjecture requires a significantly more thorough evaluation. Additional traces need to be collected in order to increase certainty that the simple classifiers we implemented significantly increase device sleep time, while maintaining event of interest recall. The classifiers we constructed may be over-fitting to the trace used in the simulations. Moreover, simple classifiers would need to be constructed for a wider-range of accelerometer-based applications.

5. CONCLUSION

TODO: most of this needs to be changed

In this paper we proposed Smartsensors, an approach that uses a low-power processor to perform sensor data acquisition and wakes up the main processor when an event of interest occurs. The API provided to developers hides the heterogeneous nature of the system, ensuring application development does not increase in complexity. The use of an API also allows Smartsensor-based mobile applications to be portable to other Smartsensor-enabled devices.

Our immediate future work includes developing Smartsensors based on other sensors available on mobile devices, such as the microphone, camera or location sensor. We believe application developers may face challenges in selecting the optimal data filter and wake-up parameters for their application. We plan on creating self-learning algorithms that, given feedback from the application about events of interest, will be able to determine the optimal data filter and wake-up parameters. Additionally, we will be exploring possible optimizations for scenarios where multiple applications are interested in similar sensor data. We are also interested in using sensor fusion to create even "smarter" sensors that enable the development of context-awake mobile applications.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] M. Kangas, A. Konttila, P. Lindgren, I. Winblad, and T. Jämsä. Comparison of low-complexity fall

detection algorithms for body attached accelerometers. *Gait & posture*, 28(2):285–291, 2008.

[10] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

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

[12] 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.

[13] 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.

[14] 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.

[15] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167):187–217, 1980.

[16] P. M. Mather. *Computer Processing of Remotely-Sensed Images: An Introduction.* Wiley, 2005.

[17] 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.

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

[19] 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.

[20] 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.