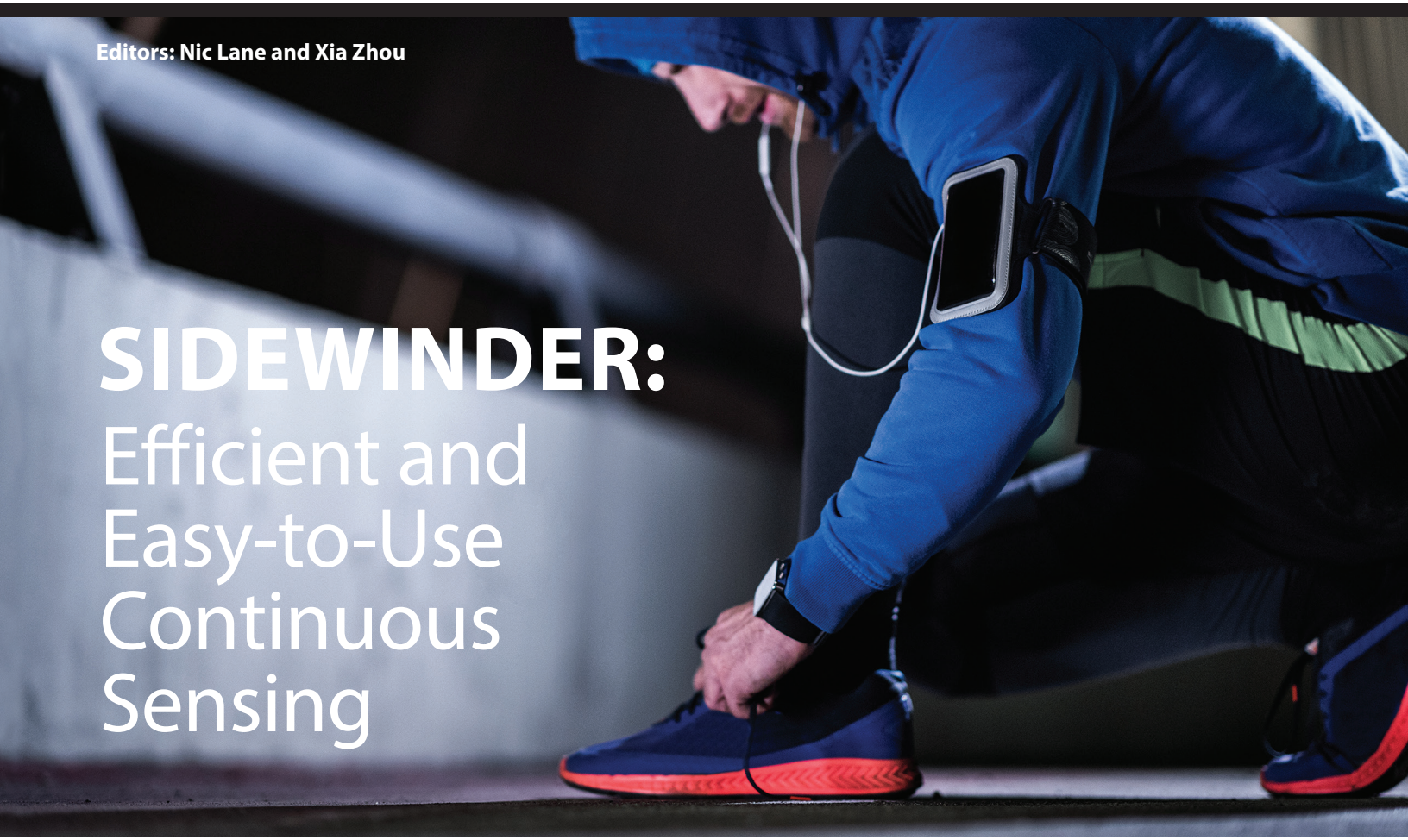
Daniyal Liaqat, Silviu Jingoi, Wilson To, Ashvin Goel and Eyal de Lara *University of Toronto*

Editors: Nic Lane and Xia Zhou



SIDEWINDER: Efficient and Easy-to-Use Continuous Sensing

Excerpted from "Sidewinder: An Energy Efficient and Developer Friendly Heterogeneous Architecture for Continuous Mobile Sensing," from *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* with permission. http://dl.acm.org/citation.cfm?id=2872398 © ACM 2016

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. We present Sidewinder, a new approach toward offloading sensor data processing to a low-power processor and waking up the main processor when events of interest occur. Sidewinder differs from other heterogeneous architectures in that developers are presented with a programming interface that lets them construct custom wake-up conditions by linking together and parameterizing predefined sensor data processing algorithms. Sidewinder's wake-up conditions achieve energy efficiency matching fully programmable offloading, but do so with a much simpler programming interface that facilitates deployment and portability.

INTRODUCTION

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. Currently, smartphones are used intermittently, where 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 down. Taking advantage of this usage pattern, mobile platforms have been heavily optimized to rely on a low-power sleep state when not in use to improve battery life.

While this approach to improving battery life has fared well with the intermittent usage scenario, it falls short in the face of continuous sensing applications, which operate on a constant stream of sensor data. These applications prevent the processor from entering its low-power sleep state, resulting in poor battery life and ultimately, a slow emergence of continuous sensing applications. Despite poor battery life, some continuous sensing applications such as Pokémon Go [1] have gained tremendous popularity. Pokémon Go players have quickly

Photo, istockphoto.com

realized that playing will drain their battery in mere hours, and many have taken to carrying around external battery packs to keep their phones alive for longer. We believe that there must be a more elegant solution.

It is generally accepted that the solution to energy-efficient continuous sensing lies in a heterogeneous architecture, in which one or more low-power, peripheral processors (referred to as a sensor hub) collect and process sensor data while the main processor is in its sleep state. The sensor hub filters out data that is unlikely to be interesting and wakes up the main processor when a potential event of interest is detected, allowing for further processing on the main processor. However, the programming model for these heterogeneous architectures remains an open research question. On one end of the spectrum, researchers have proposed a ***fully programmable*** model [2,3,4] 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 not only have to be familiar with programming the low-power processor, they also have to account for hardware differences between devices. It is also unclear how this model supports applications running concurrently. On the other end is ***predefined activity***, an approach in which hardware manufacturers provide a sensor hub that is hardwired to detect a few specific events. Both iOS and Android provide frameworks for detecting predefined activities, such as significant motion and steps [5,6]. Such frameworks are very easy to use from a developer's point of view and provide significant energy savings; however, they are limited to detecting only events predefined by the manufacturer.

Sidewinder is 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 natively on a low-power sensor hub, and application developers construct custom wake-up conditions by linking together and parameterizing these algorithms. Custom wake-up conditions execute on the low-power sensor hub and, when potential events of interest are detected, the main processor is woken up and the rest of the application code is invoked. These wake-up conditions do not need to have high precision. In fact, we envision that most wake-up conditions will have high recall and moderate precision. They will filter out the majority of sensor data, which is unlikely to be interesting, and wake up the main processor when a potentially interesting event has occurred, allowing the application to run higher precision classifiers on the main processor.

Sidewinder's design has many benefits. Firstly, it has **lower programming complexity** because application developers can use the predefined processing algorithms, rather than implementing their own. Additionally, Sidewinder allows developers to write wake-up conditions in the same language as the rest of their application, making writing wake-up conditions even easier. Secondly, Sidewinder's sensor processing algorithms can be **better optimized** because they are written and tested by experts working for the manufacturer. Sidewinder also has **better security**. Since algorithms are provided by the manufacturer, they can be trusted and reasoned about. Finally, Sidewinder wake-up conditions are **more portable**. 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) and developers do not need to write their wake-up conditions specifically for the hardware.

DESIGN

Figure 1 shows the architecture of a Sidewinder system. The sensor manager contains an API that developers can use to create their wake-up conditions. Applications interact with the sensor manager to deploy their custom wake-up conditions to the low-power sensor hub. The low-power sensor hub contains implementations of commonly used algorithms. Once configured by the developer, the wake-up condition is

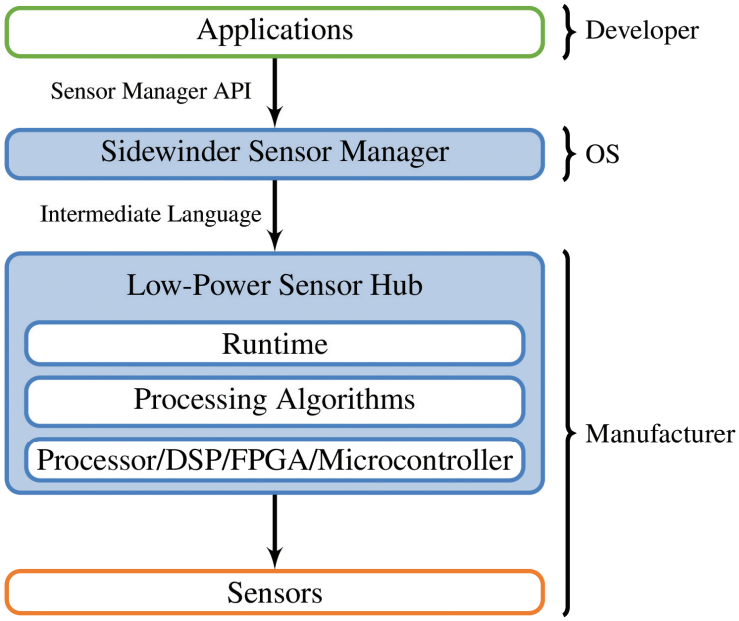


FIGURE 1. System architecture: The sensor manager is part of the OS, and the Sensor Hub and Sensors are hardware provided by the manufacturer.

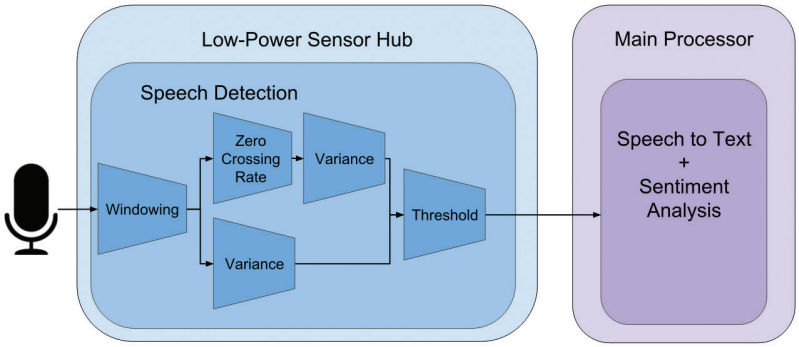


FIGURE 2. A Mood Log application in which the low-power sensor hub wakes up the main processor when speech is detected. Speech detection is performed using a relatively simple zero crossing rate method and the main processor runs more advanced speech to text and sentiment analysis.

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.

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 the sensor manager, configure the hardware and algorithms, execute the resulting wake-up condition and notify applications when an event of interest occurs.

Application developers create custom wake-up conditions for their event(s) of interest. Since common sensor data processing algorithms are provided by the manufacturer, developers only need to parameterize and chain together these algorithms to create their wake-up conditions. For example, a developer could create a mood log application that tracks the user's mood over time by converting user's speech to text and running sentiment analysis. This can be broken down into two steps, the first is determining if microphone data contains speech. If it contains speech, then the next step will convert the speech to text and run sentiment analysis. The first stage can run on the low-power sensor hub, waking the main processor only when speech is detected. The speech detection on the low-power sensor may be implemented as shown in Figure 2. Microphone data is first segmented into windows. For each window, variance of the zero crossing rate and variance of the signal amplitude is calculated. If the zero crossing rate is less than a threshold and the amplitude exceeds a threshold, the wake-up condition is satisfied and the main processor is woken up.

PROTOTYPE

Our Sidewinder 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. The Nexus 4 and microcontroller communicate over the UART port made available by the Nexus 4 debugging interface via the audio interface jack.

We implemented a sensor manager and intermediate language as part of our prototype. The Sidewinder sensor manager is based on the Android Sensor Manager [7]. It contains information about the available sensors and processing algorithms and provides developers with an API to configure custom wake-up conditions. Upon receiving a wake-up condition configuration, the sensor manager generates intermediate code that will be pushed to the hardware.

Our implementation of the Sidewinder runtime is written in C and resembles a simple interpreter. It contains implementations of Fourier Transforms, averaging, low/high pass filtering, vector magnitude, zero crossing rate and threshold algorithms.

We developed six applications to perform continuous sensing on accelerometer or microphone data. For each of the applications, we constructed a wake-up condition using the algorithms that were available on the sensor hub.

EVALUATION AND RESULTS

We evaluated Sidewinder using 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.

In our testing, we found that the Nexus 4 phone consumes 323 mW of power while awake with the screen, Wi-Fi and GPS turned off. In the sleep state, it consumes 9.7 mW of power. Additionally, the phone consumes more power during transitions between asleep and awake states. These transitional states were also accounted for in our simulation. We found the TI MSP430 microcontroller to consume 3.6 mW of power.

To obtain accelerometer traces labeled with reliable ground truth, we mounted the smartphone and microcontroller on the back of an ERA 210 robotic dog (see Figure 3), which was instructed to perform a circuit

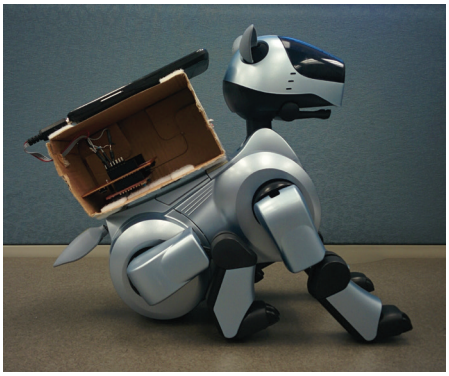


FIGURE 3. Aibo robotic dog used for data collection.

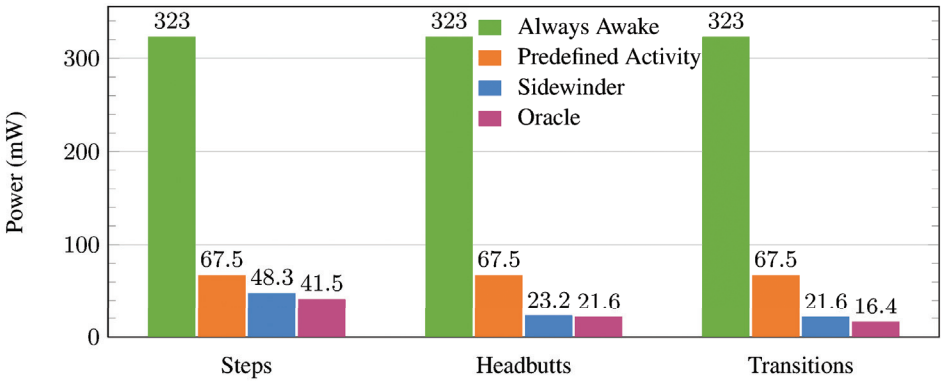


FIGURE 4. Power usage for accelerometer applications.

consisting of Steps, Posture Transitions, and Headbutts. 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.

Figure 4 shows the power usage for the accelerometer apps under four configurations. Our audio applications followed a similar trend and are not shown here. The four configurations are:

- **Always awake:** A naïve approach where the phone is always awake to process sensor data.
- **Predefined Activity:** Based on Android's built-in significant motion and a similar significant audio detector, which wake up and call an application-provided callback when acceleration or sound exceed a fixed threshold. We set the threshold so that it optimizes performance across all applications.
- **Sidewinder:** Wake-up conditions for each application were constructed by parameterizing and chaining together algorithms provided by the sensor hub.
- **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.

How Close to Optimal is Sidewinder?

By comparing the performance of Sidewinder to Oracle, we found that Sidewinder achieves between 93% and 96% of the possible power savings for our accelerometer-based applications. Audio applications performed similarly, saving between 85% and 98%. Overall, *Sidewinder realizes 85% to 98% of the potential energy savings*. This suggests that there is little additional energy savings to be realized by offering full programmability.

Sidewinder vs. Predefined Activity

In our experiments, Steps were much more likely to occur than Transitions, and Transitions were much more likely to occur than Headbutts. Since Predefined Activity cannot be parameterized in any way, it wakes up the main processor when any one of the three events occur. These false positives waste energy, especially when the event of interest occurs infrequently. We conclude that *predefined activities are unlikely to efficiently support a wide range of applications. This is particularly the case for applications interested in infrequent events.*

CONCLUSION

Sidewinder is a new approach for continuous mobile sensing. In Sidewinder, 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 chaining together and parameterizing predefined processing algorithms. We implemented a prototype based on a Nexus 4 and TI MSP430 and developed several applications using accelerometer and microphone data. We found that Sidewinder wake-up conditions were easy to write and achieved 85% to 98% of the energy savings of a theoretically optimal solution. ■

Daniyal Liaqat joined the System and Networking group at University of Toronto as a PhD student in 2014 after completing his BSc in Computing Science from Simon Fraser University. His research interests include systems level support for continuous sensing in mobile devices and applications of continuous sensing, particularly towards health monitoring.

Silviu Jingoi is a software engineer at Cisco Systems, contributing to the next-generation Software Defined Networking (SDN) strategy. He completed his MSc degree in Computer Science at University of Toronto in 2014. His main areas of interest are efficient networking systems and mobile computing, primarily conducting research on energy-efficient sensing on mobile devices.

Ashvin Goel received a Ph.D. degree in computer science and engineering from Oregon Graduate Institute, Portland in 2003, where his research focused on providing system support for interactive media-streaming applications. He is now an associate professor in the Department of Electrical and Computer Engineering at University of Toronto. His research interests are in the general area of operating systems, focusing on improving the reliability and security of software systems. He has published in systems conferences, such as SOSP, OSDI, ASPLOS, FAST and Eurosys.

Eyal de Lara is a professor in the Department of Computer Science of the University of Toronto. His research interests lie in systems-level support for mobile and cloud computing. He has a PhD in electrical and computer engineering from Rice University in Houston.

REFERENCES

[1] Pokémon Go. http://www.pokemongo.com/en-us/.
[2] 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.
[3] B. Priyantha, D. Lymberopoulos, and J. Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. Pervasive Computing, IEEE, 10(2):12–15, 2011.
[4] 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. http://developer.android.com/guide/topics/sensors/sensors_motion.html.
[5] Core motion framework reference. https://developer.apple.com/library/ios/documentation/coremotion/reference/coremotion_reference/index.html.
[6] Android motion sensors.
[7] SensorManager | Android developers. https://developer.android.com/reference/android/hardware/SensorManager.html.