Delft University of Technology
Master's Thesis in Embedded Systems

Batch Scheduling for Energy-Efficient Sensing in Smartphones

Kimon Tsitsikas

Batch Scheduling for Energy-Efficient Sensing in Smartphones

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Kimon Tsitsikas
K.TSITSIKAS@tudelft.nl

31st July 2013

Author
Kimon Tsitsikas (K.TSITSIKAS@tudelft.nl)
Title
Batch Scheduling for Energy-Efficient Sensing in Smartphones
MSc presentation
14th August 2013

Graduation Committee

Prof.Dr. Koen Langendoen (Chair)	Delft University of Technology
Dr. Matthijs Spaan	Delft University of Technology
Steven Mulder, MSc.	Sense Observation Systems
Yunus Durmus, MSc.	Delft University of Technology

Abstract

Sensing in smartphones consumes a significant amount of energy and leads to quick depletion of the battery. Most of the existing solutions to overcome the short battery lifetime caused by periodic sensing are personalized. They tend to learn and predict the user activities. Thus, fewer samples are required to recognize user state and sensing intervals can be extended. However, such methods require a training phase and any change in the user pattern causes a need for a new training phase. Therefore, in addition to personalized learning methods, we also need user-agnostic techniques that guarantee instant energy savings independently of the context to be recognized. In this thesis a user-agnostic method that seeks to provide energy efficiency in sensing is proposed. Our approach is based on the observation that an energy overhead occurs every time the CPU is woken up to perform a sensor sampling task. Hence, our goal is to decrease the number of CPU wake-ups incurred due to periodic sampling by combining multiple sensing actions into one joint activity. Our contribution is a mechanism that batches the execution of periodic tasks. BASS (BAtch Scheduler for Sensing) uses the greatest common divisor of the time intervals defined for the sensor sampling tasks. It also introduces a flexibility factor that implies the time delay tolerance regarding the execution of a task. Moreover, our tool implements a detection method for CPU wake-ups caused by any other application or the user. Based on the above, BASS applies batch scheduling to execute the sensor sampling tasks in batches and result in fewer CPU wake-ups. We evaluated our mechanism using a sensing application for monitoring patients that suffer from Rheumatic Arthritis. We conducted a number of experiments on an HTC Sensation phone, which showed that the efficient exploit of CPU wake-ups cuts down the energy consumption in mobile sensing. The BASS tool achieved an average power reduction of up to 44% and 18% in laboratory and real-world experiments respectively, in our application scenario, without compromising sensing time accuracy.

Preface

This Master of Science thesis describes my work in the Embedded Software group and Sense Observation Systems during the last eight months. My interest in Android development, as well as the hands-on experience in energy optimization that I gained throughout my Master studies inspired me to come up with this topic. The results of that work you now have before you.

I would like to thank a number of people for supporting me during this project. First of all, I would like to thank my family for their continuous support all these years. Further, I would like to thank Prof. Koen Langendoen for accepting me as a master student in the Embedded Software group, and Yunus Durmus and Ertan Onur for their guidance and advice. Finally, I would like to thank Steven Mulder, Pim Nijdam, Niels Brouwers and Andreas Loukas for their willingness to help and the great insight that they have given me.

Kimon Tsitsikas

Delft, The Netherlands
31st July 2013

Contents

Chapter 1

Introduction

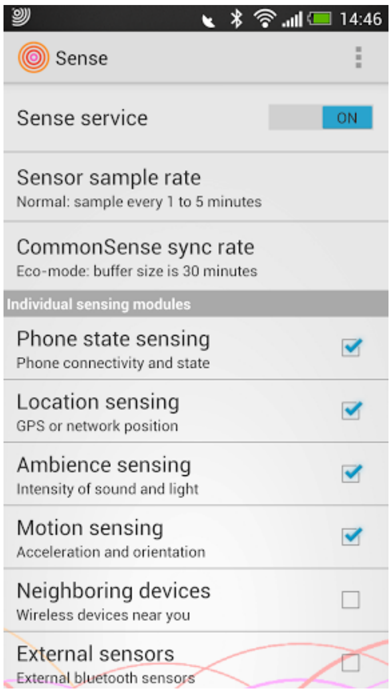
Smartphones are enhanced with a number of sensors, including accelerometer, Global Positioning System (GPS), microphone, Wi-Fi, Bluetooth etc. Sensing provides phone users the ability to capture their context, recognize their state and extract information regarding their current environment. Context awareness allows the smartphone to become more adaptive to situational changes, such as to adjust the screen brightness and ring tone volume to the appropriate level with respect to a bright or dark, respectively in a noisy or quiet place. Moreover, the contextual data could be useful to several applications and services. An indicative example is a healthcare application that helps people who suffer from depression to gather information about their mood. Gathered data then assist clinicians in guiding them.

A project that seeks to facilitate the collection of data using the sensors of mobile devices is the Sense Platform [8] (the Sense Platform user interface is shown in Figure 1.1), developed at the company Sense Observation Systems [7], where this thesis project was carried out. In this platform, data are processed (either on the device or in the cloud) to produce meaningful information (state sensors) about the person wearing the device and the person's environment. Some examples of detectable states are the following:

- Activity of a person (e.g., walking, riding a bike, sleeping, working, attending a meeting)
- Location of a person (e.g., at home, work, gym)
- Social context (e.g., crowded or quiet place)

Sense is especially interested in real-time detection of the above states. However, in order to collect this information, continuous usage of the phone CPU and sensors is required. As an example, the battery of a Nokia N95 is drained in less than 20 hours [18]. Obviously capturing data over a long period of time is hard.

Most of the existing solutions require personalization relying on training data sets [19] or a learning phase [29]. In both cases, Markov Models are


14:46

Sense

Sense service ON

Sensor sample rate
Normal: sample every 1 to 5 minutes

CommonSense sync rate
Eco-mode: buffer size is 30 minutes

Individual sensing modules

Phone state sensing ✓
Phone connectivity and state

Location sensing ✓
GPS or network position

Ambience sensing ✓
Intensity of sound and light

Motion sensing ✓
Acceleration and orientation

Neighboring devices ☐
Wireless devices near you

External sensors ☐
External bluetooth sensors

Figure 1.1: Sense Platform GUI.

trained to generate statistical chains used to predict the next user state. Prediction allows reduction in sensor sampling rates which means that less energy is consumed. In addition to personalization, the majority of the energy saving methods are specific sensor and context oriented e.g., [24] targets the energy-efficient usage of GPS in position tracking. If the user behaviour suggests repetitiveness, personalized methods can achieve energy savings. On the other hand, changes in user pattern require a new training phase for the Markov Model. During this period the method is ineffective and may result in faulty predictions. We advocate the need for application and user-agnostic methods that can also be used as a complement to prediction-based methods. Such a method guarantees energy conservation, independently of the context to be recognized.

This thesis proposes a smart sensing scheduler that aims to reduce the number of CPU wake-ups. In order to capture a sensor sample, the CPU has to wake up and process the sample data. The CPU power state transitions – from the sleep to the busy state and vice versa – imply an energy cost [27] (illustrated in Figure 1.2). According to Min et al. the energy consumption during these transitions is usually even higher than the CPU busy state power consumption [22]. Thus, periodic sensor sampling incurs an energy overhead every time that the CPU is woken up to perform the sampling process. Our goal is to eliminate this overhead by scheduling the various

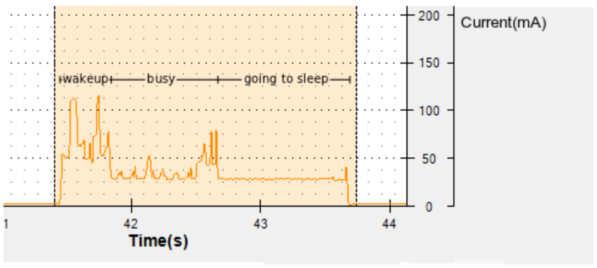


Figure 1.2: Transitions between different CPU states when capturing an accelerometer sample.

sampling tasks in batches. This will result in fewer CPU wake-ups. To this end, we also exploit the delay tolerance regarding the time of execution of a sampling task. In particular, we create a scheduling mechanism that handles the smartphone sensing requirements with respect to the balance between energy conservation and sensor sampling time accuracy.

Batch scheduling [26] is the process of grouping a number of tasks into batches before executing them on a computer. Solutions based on batch scheduling have been applied in various fields such as production and transportation [28] and parallel processing [23]. In the first case the purpose is to provide a schedule of production and transportation that minimizes the total completion time and total processing cost, while in the second case the aim is to increase the throughput. As shown in Figure 1.3 the general idea is that, given a set of sensor sampling tasks and the sensing parameters (e.g. sampling rate), BASS (BAtch Scheduler for Sensing) generates a sensing schedule of batched tasks.

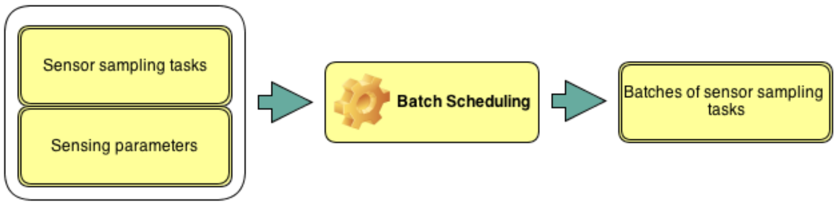


Figure 1.3: Batch scheduling applied by BASS.

To analyse the performance of our solution, new software modules were merged with the Sense Android library [6]. The Monsoon power monitor [4] and Android logging system LogCat [2] were used throughout the evaluation procedure.

1.1 Problem Statement

The main goal of this thesis is to provide an application and user-agnostic energy saving method that eliminates the CPU wake-up overhead due to the periodic sensing on smartphones. BASS seeks to reduce the number of CPU wake-ups in such a way as to capture the most accurate sensor data (in terms of time), without draining the device battery in a short time. Our sampling strategy applies batch scheduling on the various embedded sensors and also exploits the CPU wake-ups caused by other applications. Our solution aims to fulfil the requirements below:

Applicability Our mechanism has to guarantee energy conservation independently of the subset of sensors that are used and the context or state that is to be recognized. Furthermore, the energy savings have to be instant and continuous regardless of possible patterns or changes in user behaviour.

Flexibility Considering the miscellaneous sensing applications, different levels of accuracy in the time of execution of a sensor sampling task are demanded and thus the margin for energy savings varies. An ideal mechanism should be flexible in handling the trade off between the time of execution accuracy and energy consumption. In this thesis we define as *flexibility* the time delay tolerance regarding the execution of a task.

1.2 System Overview

The inputs into our system are a set of sensor sampling tasks, the corresponding time intervals and flexibilities. Based on this data, our objective is to schedule the execution of the tasks as to minimize the number of CPU wake-ups and thus to decrease the energy consumed. To this end, BASS applies batch scheduling in three ways:

1. Initially BASS groups the tasks based on the greatest common divisor (gcd) of their time intervals.

2. Taking into consideration the flexibility, BASS postpones their time of execution in order to overlap each other.

3. Finally, BASS opportunistically executes a task when it detects that the CPU is woken up to perform a task for another application.

The above techniques are discussed in detail in Chapter 4.

1.3 Organization

This thesis is organized as follows. Chapter 2 consists of more background information regarding CPU wake-up energy overhead and batch scheduling. In Chapter 3 the overall design of BASS is presented. The detailed description of its implementation is included in Chapter 4. The results from experiments conducted are discussed in Chapter 5. Finally, conclusions and future work are given in Chapter 6.

Chapter 2

Background and Related Work

The purpose of this chapter is to provide the background needed for BASS, as well as to discuss the related work. In Section 2.1 we describe the CPU power states and the process of CPU wake-up while the motivation behind our approach is also presented. Finally, in Section 2.2 related work is discussed.

2.1 CPU Energy Consumption

Energy consumption due to CPU usage in computational devices (such as smartphones) can be described using indicative terms regarding the different CPU states and transitions between them. These states are defined based on the CPU power states (see Subsection 2.1.1). A transition of special interest with regard to energy consumption is the one when the CPU wakes up (see Subsection 2.1.2). Since the smartphones used in this thesis use the Linux-based Android operating system [3], the following information concerns Android type of platforms.

2.1.1 Power States

The Advanced Configuration Power Interface (ACPI) defines the degree to which the processor is "sleeping" by specifying core power states known as the CPU C-states. Table 2.1 provides an overview of the main C-states. C0 indicates that the processor is actively running code while all other C-states (C1-Cn) describe states where the processor clock is disabled and various parts of the processor are shut-off [20].

While a processor operates (i.e. the CPU is in the C0 state), it can be in one of several CPU performance states (P-states). P-states are operational states that relate to CPU frequency and voltage. Drivers known as CPU

C-State	Description
C0	CPU fully turned on
C1	Stops CPU main internal clocks via software
C1E	Stops CPU main internal clocks via software and reduces CPU voltage
C2 & up	Stops CPU main internal clocks via hardware

Table 2.1: CPU C-states.

governors use P-states to raise and lower the frequency in response to the demands of the applications running [25]. Changes in the frequency allow to lower the voltage and thus impact energy consumption in a squared manner. This is modelled by the CPU power consumption formula $P = CV^2f$, where C is capacitance, f is frequency and V is voltage.

In this thesis, the CPU power consumption is expressed using the convention of two representative states: The *busy state* where the CPU is executing some task (i.e., the CPU is in the C0 state) and the *sleep state* where the phone is in deep-sleep mode and barely consumes energy (i.e., the CPU is in C1 or a higher state). We also define the transitions from the sleep to busy and from busy to sleep state as *wake-up* and *going-to-sleep*, respectively. Whenever the system is woken up by an interrupt, it enables the interrupt handler and other hardware components, wakes up the CPU and resumes past processes [14]. In particular, during the wake-up transition, a mechanism that keeps the CPU on – the wakelock [9] – is acquired and released right after the completion of the operation that is to be performed. In case there are no wakelocks acquired, the system freezes all processes and suspends the CPU and hardware components. The states and transitions that we described are illustrated in Figure 2.1.

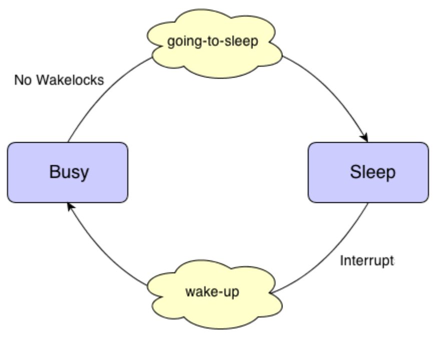


Figure 2.1: CPU power states and transitions.

2.1.2 Wake-up Overhead

The wake-up transition implies extra energy cost. The reason is that waking up from deeper CPU sleep states requires longer times in higher frequency states [20]. Thus according to the CPU power consumption formula an energy overhead is induced. Taking this into consideration, the energy consumed in order to perform a task (e.g., process of sensory data) is described by defining the following three attributes:

- E_w: Energy spent in wake-up transition.
- E_s: Energy spent in going-to-sleep transition.
- E_b: Energy spent in busy state.

Figure 2.2 illustrates the above energy costs.

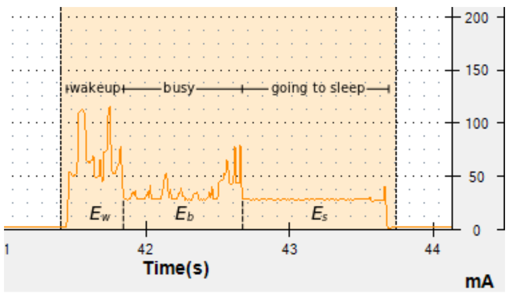


Figure 2.2: Energy spent in different power states.

We also conventionally define the CPU energy overhead E_o as the sum of wake-up and going-to-sleep transition energy costs:

$$E_o = E_w + E_s \tag{2.1}$$

From the above it is deduced that for the execution of N tasks $E_{n_{worst}}$ energy is consumed:

$$E_{n_{worst}} = N(E_b + E_o) \tag{2.2}$$

It follows that, if CPU is able to merge all the N tasks in one joint task, the energy consumed is

$$E_{n_{best}} = NE_b + E_o. \tag{2.3}$$

If we assume that K out of N tasks coincide and thus only one wake-up is required for each of them, the number of wake-ups L becomes $L = (N - K + 1)$ and energy consumption is

$$E_n = NE_b + LE_o. \tag{2.4}$$

Most smartphone applications require their tasks to be executed periodically, with a certain time interval between each execution. In the case of

Sense Platform, each sensor samples in accord with a time interval specified by the user. Thus, if N different sensors are about to sample and K of these samples coincide the consumed energy can be described by (2.4). An ideal energy-aware scheduling of these sampling tasks maximizes the coincidences in order to minimize the energy consumption. However, the time interval for each sensor can be different and therefore an optimized schedule is needed.

In this thesis it is also taken into consideration the fact that in the meantime the CPU may have been woken-up by some other application or the user. Brouwers et al. describe a technique with which it is possible to detect when the CPU is awakened by some application [12]. Specifically, because of the way that the $Thread.sleep()$ method is implemented in Android, the timers on which the sleeping behaviour relies are frozen once the processor is in the sleep state. Hence, a sleeping thread will continue its execution once CPU is woken up by another process. In this thesis we are motivated by this knowledge. We make our scheduler aware of CPU wake-ups caused by any other application and able to opportunistically execute tasks in order to prevent a CPU wake-up later on. However, to achieve such a behaviour, we exploit the different options provided by the Alarm Manager [1] of Android (the exact methodology is described in Chapter 4).

2.2 Related Work

In this section we present the current state of the art with respect to energy efficient sensing in smartphones. The existing solutions are categorized as *prediction-based* (Subsection 2.2.1) and *specific-sensor oriented* (Subsection 2.2.2). We also discuss the related work regarding the three different aspects of our scheduling mechanism as they are enumerated in Chapter 1 (see Subsection 2.2.3).

2.2.1 Prediction-based Methods

Prediction-based approaches for energy conservation require personalization. They recognize the habitual behaviour of the user. In this way they can predict his next state and decrease the sampling rates. Such a solution is proposed by Wang et al. Continuing their previous work [30], the authors describe an approach to estimate the most likely user state when sensor observations are missing [29]. Their solution is based on the assumption that user state evolves as a discrete time semi-Markov process. The time interval between each observation is adjusted so as to minimize the expected state estimation error while maintaining an energy consumption budget. Their approach however is admissible to criticism. Wang et al. consider the sense observations to be perfect which is unlikely due to factors such as location provider inaccuracy or noise in sound sensing. Another disadvantage is that the mechanism provided relies on a learning phase which is conducted once.

This renders the estimation policy maladaptive to changes in user behaviour. Furthermore, a communication overhead is required. The calculation of state estimation takes place in a server.

Gordon et al. present a methodology that matches sensor configurations to activities (the number of sensors depends on an acceptable loss parameter) [19]. The authors use a first-order Markov chain to predict the next activity and switch on/off the corresponding sensors. However, the prediction algorithm lacks adaptiveness as it depends on the initially used training data set. Furthermore, the same and constant sampling rate is defined for all the sensors.

2.2.2 Specific-Sensor Oriented Approaches

Specific-sensor oriented approaches target the energy-efficient usage of particular sensors. Paek et al. describe a tool developed based on a collection of techniques for efficient use of GPS [24]. In general, the location-time history of the user is exploited in order to eliminate the use of GPS in case it is unproductive. This approach focuses on energy conservation only from the perspective of limited GPS usage. The accelerometer is duty-cycled in an empirical way. The mechanism proposed in [15] schedules the use of the different location providers given an average localization error. In addition, prediction opportunities are exploited by generating a logical mobility tree that includes uncertainty points (e.g. traffic intersections) after which a location reading is necessary. A drawback of this solution is that the usage of a linear predictor may be unsuitable in case the phone's movement is not straight. Moreover, according to the authors, varying speeds or pauses in user mobility cause the prediction to be imprecise.

Besides the location-oriented approaches, another method describes an activity-adaptive accelerometer for activity recognition [31]. Depending in the identified current activity, the algorithm determines the most efficient – both in terms of accuracy and energy consumption – combination of sampling frequency and classification features (e.g. mean, variance, etc). However, the energy cost of the accelerometer is rather trivial compared to other sensors (see Chapter 3). This solution is ineffective in case of an application that uses multiple sensors.

2.2.3 Batch Scheduling Techniques

BASS applies batch scheduling using three different features: (1) The greatest common divisor (gcd) of the sampling time intervals, (2) the flexibility regarding the time of execution of the sampling tasks, (3) the detection of CPU wake-ups caused by other applications. With respect to the above, we discuss related work in various fields other than smartphone sensing:

Greatest common divisor (gcd) In an already existing approach, batch scheduling is applied in order to accumulate the clock interrupts needed to be sent to a Virtual Machine (VM) [10]. The proposed algorithm introduces a batching factor N and, given the VM time period T, dispatches N interrupts every $T \times N$. Although such a batch scheduling is efficient for a fixed time interval such as in the case of time period T, it is rather rigid for interrupts occurred in variable time intervals (i.e., periodic sensor samples). Another batch scheduling technique addresses the case of recurrent applications [13]. The gcd of the different application time intervals is used in order to enforce the execution of applications with gcd greater than 1 to coincide and thus to have fewer CPU wake-ups. However, the algorithm described is eligible only in the case that the gcd is equal to the smallest interval. In the worst case scenario where time intervals gcd is equal to 1, it results in redundant wake-ups. Such an approach is inadequate in our case as it overlooks some arbitrary events (e.g. CPU wake-up caused by the user or another application) and possible time delay tolerance in the execution of a task.

Flexibility Similarly to our flexibility, Davis et al. make use of slack time in pre-emptive systems scheduling [16]. However, in contrast to our BASS where the objective is to batch the execution of the tasks, this algorithm aims to satisfy the task priorities. Another method aims to optimize performance and minimize energy consumption in heterogeneous Network-on-Chip architectures via communication and computation task scheduling [21]. A heuristic algorithm based on slack budgeting and performance constraints is proposed. While this approach exploits the multiple tiles in heterogeneous platforms, BASS provides a solution that is efficient even in single-core processor systems.

Balasubramanian et al. describe a scheduling algorithm that reduces the energy overhead of mobile networking [11]. The authors exploit the time delay tolerance regarding the transmission requests in order to defer them in such a manner that they are sequentially transmitted. This way, tail energy (i.e. energy wasted in high-power states after the completion of a transfer) is minimized. This approach seeks to eliminate the energy overhead in networking resources usage. BASS on the other hand, applies a scheduling strategy to decrease the corresponding overhead caused by the processing unit use.

CPU wake-ups detection Diao et al. [17] apply machine learning and particularly a Dynamic Bayesian Network (DBN) model to export the CPU activity pattern for a multi-core package and predict the next states in a given horizon. This algorithm aims to improve busy state duration prediction accuracy, however it incurs a prohibitive computational overhead as it requires frequent predictions. On the other hand, Min et al. provide an estimator that predicts the next CPU sleep state duration based on recent history [22]. The authors propose a mechanism to predict deterministic and stochastic interrupts that end the sleep state.

BASS elaborates and combines most of the above concepts, providing a system that integrates different energy conservation methods. Thus, a more flexible and robust scheduling mechanism is generated and implemented.

Chapter 3

Energy Analysis

This chapter provides an energy analysis of smartphone sensing. Section 3.1 presents the devices used in the experiments conducted through the evaluation procedure. Section 3.2 provides an experimental analysis regarding the energy consumed in CPU wake-up and sensor sampling.

3.1 Experimental Setup

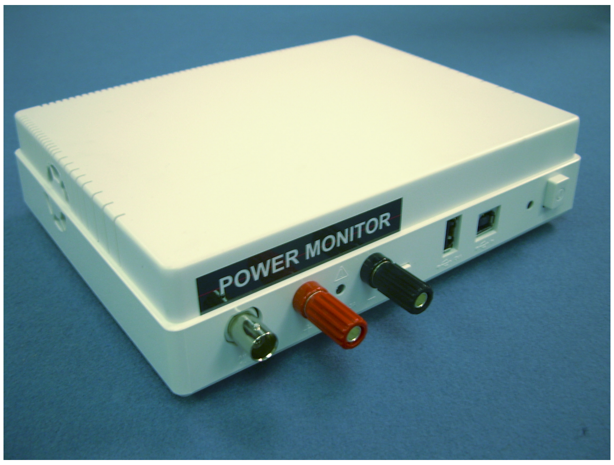


Figure 3.1: Monsoon power monitor.

In this section we describe the setup that we use for our energy analysis (see Section 3.2) and evaluation procedure (see Chapter 5). Our energy measurements are conducted using the Monsoon power monitor shown in Figure 3.1. Monsoon analyses the power consumption on any device that uses a single lithium (Li) battery, hence it can be used for energy optimization on mobile devices.

Throughout this thesis we perform measurements on three different smart-

Phone Model	Processor	Android Version
HTC Sensation	Dual-core 1.2 GHz Scorpion	v2.3.4 (Gingerbread)
Samsung Galaxy Gio	800 MHz	v2.2 (Froyo)
Samsung Galaxy S II	Dual-core 1.2 GHz Cortex-A9	v4.0.4 (Ice Cream Sandwich)

Table 3.1: Phone models used in experiments.

phones: an HTC Sensation, a Samsung Galaxy Gio and a Samsung Galaxy S II. Table 3.1 provides an overview of the phone models. We choose these particular phones because of their variety both in terms of processor type and Android version. For easy connection to the Monsoon, each phone battery is modified as shown in Figure 3.2. In particular, the battery is covered with two layers of kapton tape. Between them, there are two strips of copper tape. The one end of each strip is connected to one of the battery poles and the other to a cable that can be hooked up to the Monsoon connectors.

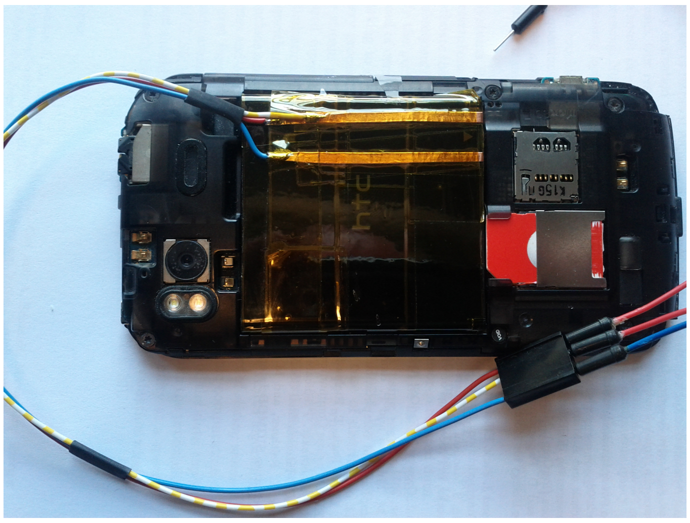


Figure 3.2: Phone modified in order to connect to power monitor.

3.2 Energy Analysis

In Chapter 2 we defined the CPU energy overhead E_o as the sum of the *wake-up* and *going-to-sleep* transition energy costs. In order to measure the energy consumed in the overhead, we wake up the CPU without assigning any task (i.e., the energy spent in busy state is trivial). Figure 3.3 illustrates the measurements conducted using three different smartphones. As it is shown, the overhead energy consumption exists and varies from phone to phone.

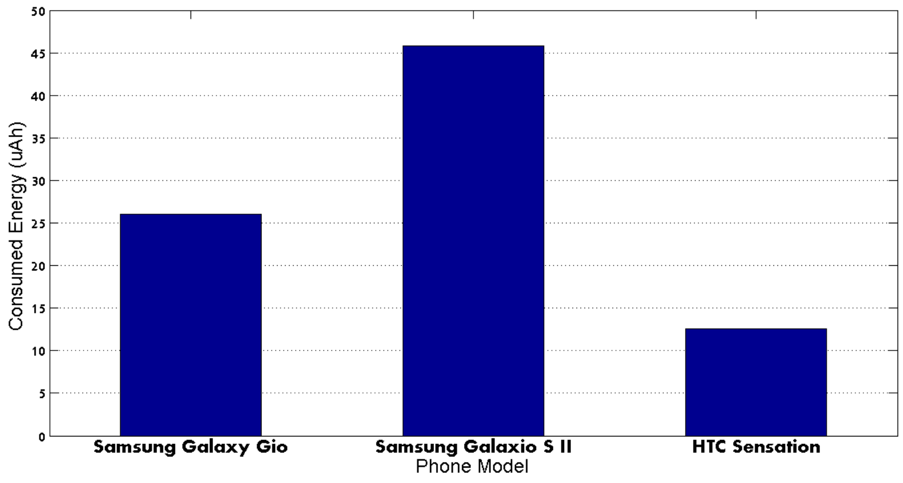


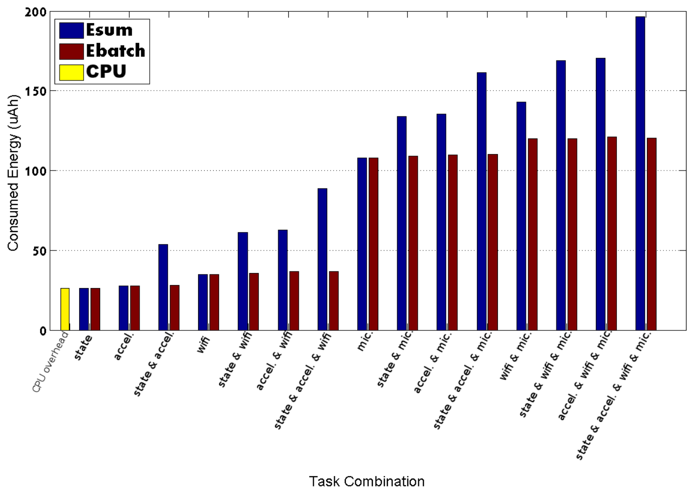
Figure 3.3: Energy consumed in CPU energy overhead.

We conventionally define four indicative sensor sampling tasks that are widely used in sensing applications, picked out from the main four classes of sensors (Phone State, Motion, Location, Ambience):

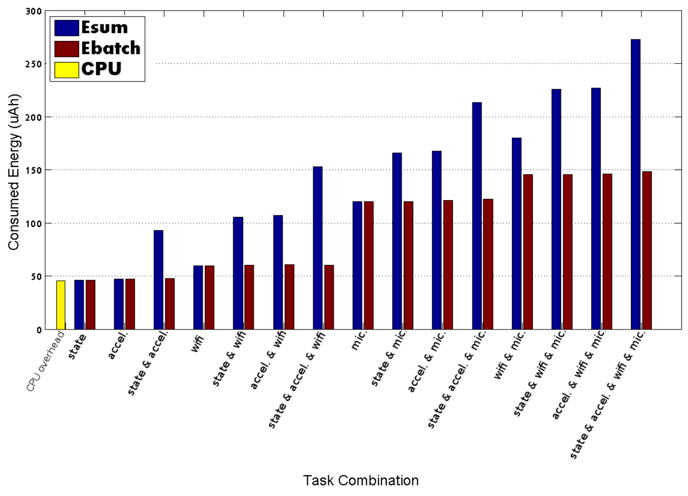
- $s_{\textbf{state}}$: Detect the call state of the phone.
- $s_{\textbf{accel.}}$: Obtain one accelerometer sample.
- $s_{\textbf{wifi}}$: Scan for single Wi-Fi channel.
- $s_{\textbf{mic.}}$: Record sound using the microphone for 5 seconds at 44100Hz frequency.

We also define the cumulative task C_S where S is the set of sensor sampling tasks that are executed in batch (e.g. $C_{\{s_{\text{state}}, s_{\text{accel.}}\}}$ is a cumulative task that both detects the call state and captures an accelerometer sample). In order to compare the energy consumption of different task configurations to the energy consumption of the corresponding cumulative tasks we define the tables E_{sum} and E_{batch} (see Table 3.2) respectively, where $E(s_n)$ symbolizes the energy consumed for the execution of s_n. Due to a technical problem occurred in HTC Sensation during this experimental session, the experiments for this comparison were conducted only on Samsung Galaxy Gio and Samsung Galaxy S II. From the measurements illustrated in Figures 3.4a and 3.4b, it derives that for each and every combination of tasks s_{state} to $s_{\text{mic.}}$ the energy required to separately execute them is more than when they are executed in a batch.

Thus, cumulating tasks guarantees energy savings for every possible combination of sensor sampling tasks. What is more, we can observe that the



(a) Samsung Galaxy Gio.



(b) Samsung Galaxy S II.

Figure 3.4: Comparison of energy consumed by different sensor sampling task combinations and corresponding cumulative tasks.

Combination Abbreviation	E_{sum}	E_{batch}
state	$E(s_{\text{state}})$	$E(C_{\{s_{\text{state}}\}})$
accel.	$E(s_{\text{accel.}})$	$E(C_{\{s_{\text{accel.}}\}})$
state & accel.	$E(s_{\text{state}}) + E(s_{\text{accel.}})$	$E(C_{\{s_{\text{state}}, s_{\text{accel.}}\}})$
wifi	$E(s_{\text{wifi}})$	$E(C_{\{s_{\text{wifi}}\}})$
state & wifi	$E(s_{\text{state}}) + E(s_{\text{wifi}})$	$E(C_{\{s_{\text{state}}, s_{\text{wifi}}\}})$
accel. & wifi	$E(s_{\text{accel.}}) + E(s_{\text{wifi}})$	$E(C_{\{s_{\text{accel.}}, s_{\text{wifi}}\}})$
state & accel. & wifi	$E(s_{\text{state}}) + E(s_{\text{accel.}}) + E(s_{\text{wifi}})$	$E(C_{\{s_{\text{state}}, s_{\text{accel.}}, s_{\text{wifi}}\}})$
mic.	$E(s_{\text{mic.}})$	$E(C_{\{s_{\text{mic.}}\}})$
state & mic.	$E(s_{\text{state}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{state}}, s_{\text{mic.}}\}})$
accel. & mic.	$E(s_{\text{accel.}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{accel.}}, s_{\text{mic.}}\}})$
state & accel. & mic.	$E(s_{\text{state}}) + E(s_{\text{accel.}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{state}}, s_{\text{accel.}}, s_{\text{mic.}}\}})$
wifi & mic.	$E(s_{\text{wifi}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{wifi}}, s_{\text{mic.}}\}})$
state & wifi & mic.	$E(s_{\text{state}}) + E(s_{\text{wifi}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{state}}, s_{\text{wifi}}, s_{\text{mic.}}\}})$
accel. & wifi & mic.	$E(s_{\text{accel.}}) + E(s_{\text{wifi}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{accel.}}, s_{\text{wifi}}, s_{\text{mic.}}\}})$
state & accel. & wifi & mic.	$E(s_{\text{state}}) + E(s_{\text{accel.}}) + E(s_{\text{wifi}}) + E(s_{\text{mic.}})$	$E(C_{\{s_{\text{state}}, s_{\text{accel.}}, s_{\text{wifi}}, s_{\text{mic.}}\}})$

Table 3.2: The energy consumption for all possible combinations of tasks s_{state} to $s_{\text{mic.}}$, when they are executed separately (E_{sum}) and batched (E_{batch}).

ratio between the energy savings for different task configurations is similar in both Samsung Galaxy Gio and S II (e.g. cumulating s_{state}, $s_{\text{accel.}}$ and s_{wifi} is in both cases the most effective combination in terms of energy savings, while cumulating s_{wifi} and $s_{\text{mic.}}$ is the least effective). Hence, if we assume that, in any given time range, a set of sensor sampling tasks have to be performed and different cumulations are possible, the optimal set of cumulations will be the same for both of the smartphones.

Based on the above results, in Chapter 4 we provide a software architecture which handles a set of tasks in such a way that corresponding cumulative tasks are generated.

Chapter 4

Batch Scheduling

This chapter provides an overview of BASS and details all the steps performed in order to reduce the number of CPU wake-ups that are occurred due to the periodic sensing. In Section 4.1 the general principles are briefly discussed while Section 4.2 describes the software architecture of BASS in more detail. The methods presented in Sections 4.3, 4.4 and 4.5 are applied on a set of sensor sampling tasks in order to generate equivalent cumulative tasks.

4.1 General Idea

Contrary to already existing solutions (see Chapter 2) in this thesis we propose an application and user agnostic approach for energy conservation in mobile sensing. BASS schedules the execution of the sensor sampling tasks in batches in order to reduce the number of CPU wake-ups. In particular, BASS seeks to detect when two or more tasks can be scheduled to overlap with each other or with the execution of another application task. The functionality performed by BASS is described by the following steps:

1. The sensing application registers to BASS its sensor sampling tasks, their time intervals and flexibility regarding their execution.

2. BASS groups the tasks in batches by superposing and shifting based on the time intervals and available flexibility.

3. BASS generates cumulative tasks and the corresponding times of execution.

Step 1 is discussed in further detail in Section 4.2 while steps 2 and 3 are thoroughly described in Sections 4.3, 4.4 and 4.5.

4.2 Software Architecture

In this section, we provide a software architecture (see Figure 4.1) which handles a set of tasks such that corresponding cumulative tasks are generated. BASS consists of three main components:

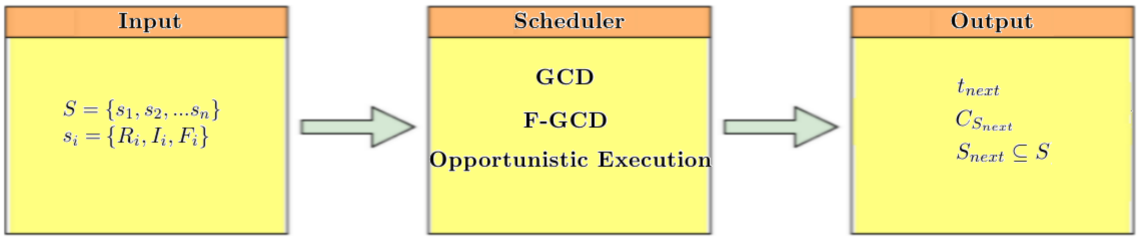


Figure 4.1: BASS Software Architecture.

Input A set of sensor sampling tasks $S = \{s_1, s_2, ...s_n\}$. Each task $s_i = \{R_i, I_i, F_i\}$ is described by a runnable R_i that operates the actual sampling process, the time interval I_i between each execution of task s_i and the flexibility F_i (i.e. the time delay margin regarding each execution of this task).

Scheduler In Chapter 2 we defined $E_n = NE_b + LE_o$ (where E_o is the energy overhead) as the energy consumed for the execution of N tasks. Given the input, our scheduler aims to schedule the times of execution of the tasks in such a way so as to minimize the number of CPU wake-ups (L) needed for the completion of the tasks.

Output The scheduler generates the next time of execution t_{next} and the corresponding cumulative task $C_{S_{next}}$ (where $S_{next} \subseteq S$) that is scheduled to be executed.

In order to achieve the above described functionality, BASS applies a number of batch scheduling techniques on the given input:

Greatest Common Divisor based Scheduling (GCD) GCD method maximizes the overlapping in the execution of the different tasks. It gathers the tasks for which the time intervals have gcd greater than 1 and schedules their executions in order to superpose.

Flexible GCD (F-GCD) In case there is another task scheduled within a task's flexibility margin, F-GCD shifts forward the second's execution and merges them in a cumulative task. The shifted task may even be an already cumulative task.

Opportunistic Execution The Opportunistic method shifts backward the execution of a task in case the CPU is woken up within its flexibility margin. Similarly to F-GCD, the shifted task may be a cumulative task.

Below we present in detail these three methods. Their performance is evaluated in Chapter 5.

4.3 Greatest Common Divisor based Scheduling (GCD)

We described the input to BASS as a set of sensor sampling tasks, where each task consists of a runnable R_i, the time interval I_i and flexibility F_i (i.e. $s_i = \{R_i, I_i, F_i\}$). In addition to these parameters, BASS introduces the *nextExecution* of a task. The *nextExecution* property refers to the time when the next execution of this task is scheduled. When a task is to be scheduled for the first time, our scheduler follows the procedure described in Algorithm 1. Initially the task whose time interval has the greatest common divisor with the new task is found. The times of execution of these two tasks are then scheduled to coincide. Thus, only one CPU wake-up is required for the execution of both of them every time that their times of execution overlap.

```
Data: new task
Result: next time of execution of new task
tasks ← get_tasks();
found_task ← null;
gcd ← 1;
foreach $tasks_i$ do
    // find the task which has gcd with the new task
    temp_gcd ← gcd(new_task.I, $tasks_i$.I);
    if $temp\_gcd > gcd$ then
        gcd ← temp_gcd;
        found_task ← $tasks_i$;
    end
end
if $gcd > 1$ then
    // executions scheduled to coincide
    new_task.nextExecution ← found_task.nextExecution;
end
else
    new_task.nextExecution ← now();
end
return new_task.nextExecution;
```

Algorithm 1: New Task Scheduling Algorithm.

The example illustrated in Figures 4.2 and 4.3 shows a possible execution of tasks $s_1 = \{A, 2, 0\}$ and $s_2 = \{B, 4, 0\}$ before and after GCD is applied. As shown in Figure 4.2, a random schedule of the two tasks can result in one CPU wake-up per task execution. One the other hand, when GCD is applied (see Figure 4.3) the first execution of task s_2 is scheduled to coincide with the execution of task s_1. Thus their executions will overlap every $\frac{s_2.I}{gcd(s_1.I, s_2.I)} = 2$ executions of s_1. As it can be seen, GCD results in reduction of the number of CPU wake-ups needed for the periodic execution of the two tasks.

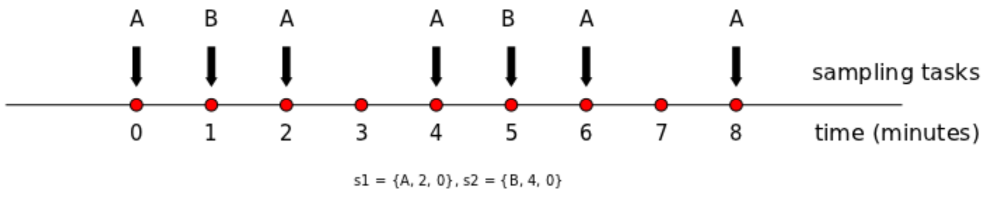


Figure 4.2: Tasks s_1 and s_2 are randomly scheduled, thus one CPU wake-up per task execution is needed.

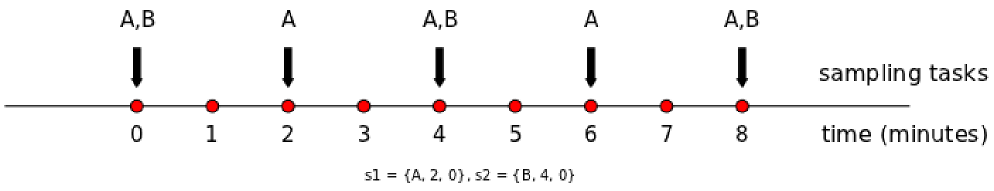


Figure 4.3: Tasks s_1 and s_2, with 2 and 4 minutes interval respectively and *gcd* equal to 2 minutes, are scheduled in order to overlap every 4 minutes.

4.4 Flexible GCD (F-GCD)

The goal of this method is to cope with the variances in sensor task time intervals. Since sensing applications may use multiple sensors with different sampling rates, it is rather unlikely that GCD is always applicable (e.g., in case of prime numbers as intervals). Flexible GCD seeks to exploit the chance of overlapping task executions even in case the *gcd* of their intervals is equal to 1. To this end, the BASS scheduler shifts forward the execution of the upcoming task every time that another task is detected within its flexibility margin and merges them in one cumulative task (see Algorithm 2). This procedure takes place even when the flexibility is equal to 0, so as to create a cumulative task out of the tasks that coincide because of GCD method. In Figure 4.4 intervals of tasks $s_1 = \{C, 5, 2\}$ and $s_2 = \{D, 6, 0\}$

have *gcd* equal to 1 and simple GCD as discussed in Section 4.3, is insufficient. However, in the case of s_1 second execution, task s_2 is found within s_1 flexibility margin and thus s_1 is shifted in order to be merged with s_2 in one cumulative task.

```
Data: sorted list of tasks in ascending order with respect to their
      next time of execution property
Result: cumulative task and its time of execution
// get the time of execution, flexibility and runnable
// of the task that is to be executed next
next_execution_time ← sorted_$tasks_0$.nextExecution;
remaining_flexibility ← sorted_$tasks_0$.F;
cumulative_task.add(sorted_$tasks_0$.R);
for i = 1, i<sizeof(sorted_tasks), i++ do
    // calculate the time difference between the two tasks
    // and deduct it from the remaining flexibility
    remaining_flexibility -= (sorted_$tasks_i$.nextExecution -
    sorted_$tasks_{i-1}$.nextExecution);
    if remaining_flexibility>=0 then
        // postpone time of execution to batch with
        // this task
        next_execution_time ← sorted_$tasks_i$.nextExecution;
        cumulative_task.add(sorted_$tasks_i$.R);
    end
    // if the flexibility of the current task is lower
    // than the remaining, update
    if sorted_$tasks_i$.F<remaining_flexibility then
        remaining_flexibility = sorted_$tasks_i$.F;
    end
end
// update the nextExecution time of the tasks that are in
// the cumulative task
for j = 0, j < i, j++ do
    sorted_$tasks_j$.nextExecution += sorted_$tasks_j$.I;
end
return (cumulative_task , next_execution_time);
```

Algorithm 2: Cumulative Task Creation Algorithm.

Note that the function that implements the above algorithm is called in two cases:

- After the last created cumulative task has been executed.
- When a new task is registered or its interval is reset. In this case,

we first recompute the *nextExecution* times of the tasks and then we schedule the next execution according to the algorithm 2. Any cumulative task that may be already scheduled is discarded.

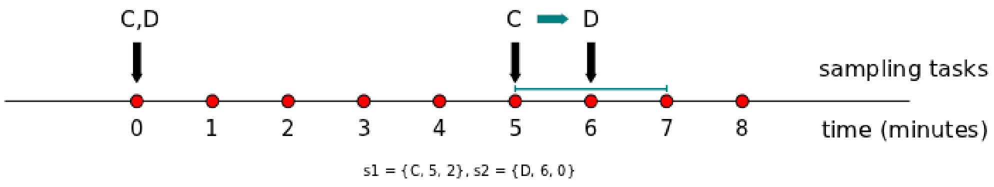


Figure 4.4: The second time of execution of tasks s_1 and s_2, with 5 and 6 minutes interval respectively, is different. However, due to the 2 minutes flexibility margin of task s_1 its execution is postponed by 1 minute in order to be batched with the execution of s_2. It is worth clarifying that the third time of execution of tasks s_1 and s_2 is the 10th and 12th minute, respectively, thus they will again be batched on the 12th minute.

4.5 Opportunistic Execution

In a real-world scenario, multiple applications are running simultaneously, causing several CPU wake-ups in order to perform their tasks. In addition, a CPU wake-up is incurred every time that the user presses a button while the CPU is in the sleep state. Opportunistic Execution seeks to detect the above wake-ups and schedule the sensor sampling tasks in a way to exploit the fact that the CPU is already awake.

The Alarm Manager of the Android platform provides the ability to schedule the execution of a task after a given time interval. What is more, we are able to differentiate the behaviour of the Alarm Manager in two ways:

1. The scheduled task will be executed even if the CPU is in the sleep state and has to be woken up.

2. The execution of the scheduled task is performed just once the CPU is woken up due to an event caused by the user or another application.

This way, our scheduler is aware of CPU wake-ups caused by other applications and is able to opportunistically execute sensor sampling tasks so as to take advantage of an already occurred CPU wake-up. In the case of Opportunistic Execution method, flexibility indicates how earlier than its next time of execution a task can be executed. The exact scheduling procedure is described in detail in Appendix 7, where the flexibility used is the remaining flexibility after the creation of the cumulative task (see Algorithm 2). The BASS scheduler shifts backward the execution of a task in case the CPU is woken up within the flexibility margin of the task, thus the need for a later CPU wake-up is eliminated (see Figure 4.5). Such as in F-GCD, the shifted task may be a cumulative task. In this case, more than one methods are combined.

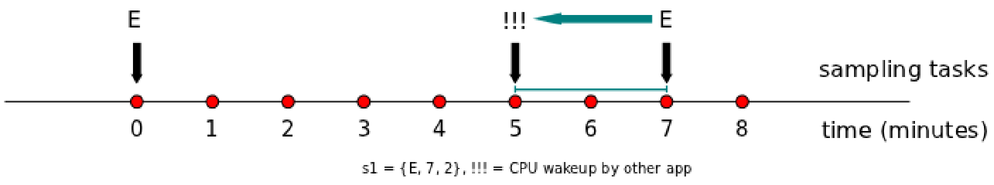


Figure 4.5: Although the second execution of task s_1 is scheduled for the 7th minute, due to the 2 minutes flexibility margin, it is performed on the 5th minute when the CPU is woken up by another application.

Chapter 5

Experimental Results

This chapter presents the experiments conducted. Section 5.1 provides the test cases under which the three batch scheduling methods described in Chapter 4 are evaluated, and presents the result of a real-world experiment. In Section 5.2, the experimental results are discussed and the problems encountered throughout the experimental procedure are given.

5.1 BASS Evaluation

As we mentioned in Chapter 1, Sense uses various sensor sampling task configurations of its Sense Platform in order to recognize different user states. One of these configurations is applied to monitoring patients that suffer from Rheumatoid Arthritis [5]. For the purpose of brevity we refer to this application using the name Rheuma App. Table 5.1 provides an overview of the sensor sampling tasks Rheuma App consists of. According to the default settings of the application, the time interval for tasks s_{state} to $s_{prox.}$ is 60 seconds while for $s_{loc.}$ it is 300 seconds.

In order to evaluate the performance of GCD, F-GCD and Opportunistic Execution, three different test cases are presented. The basis of all cases

Task	Description	I (seconds)	F (seconds)
s_{state}	Call state of the phone	60	12
$s_{accel.}$	One accelerometer sample	60	12
$s_{mic.}$	Sound from the microphone (5 sec)	60	12
s_{light}	Ambient light intensity	60	12
$s_{magn.}$	Ambient magnetic field	60	12
$s_{prox.}$	Proximity of the smartphone to nearby objects	60	12
$s_{loc.}$	Location using cell tower and Wi-Fi access points	300	60

Table 5.1: Sensor sampling tasks of Rheuma App.

is the Rheuma App sensor sampling task configuration. However, the time intervals are differentiated in the second test case. HTC Sensation and Monsoon power monitor (see Chapter 3) were used for our experiments in all of the three cases. We have to note that the Airplane mode was on and no other functionality of the phone was taking place during our measurements. It is worth clarifying that Wi-Fi can be enabled when Airplane mode is on, thus $s_{loc.}$ is still executed. Each experiment lasted for 1 hour. In each test case, the average power measurement was instantly given by the Monsoon, while the number of task executions was retrieved using the Android logs after the completion of the experiment.

5.1.1 Case 1: Best Case Scenario

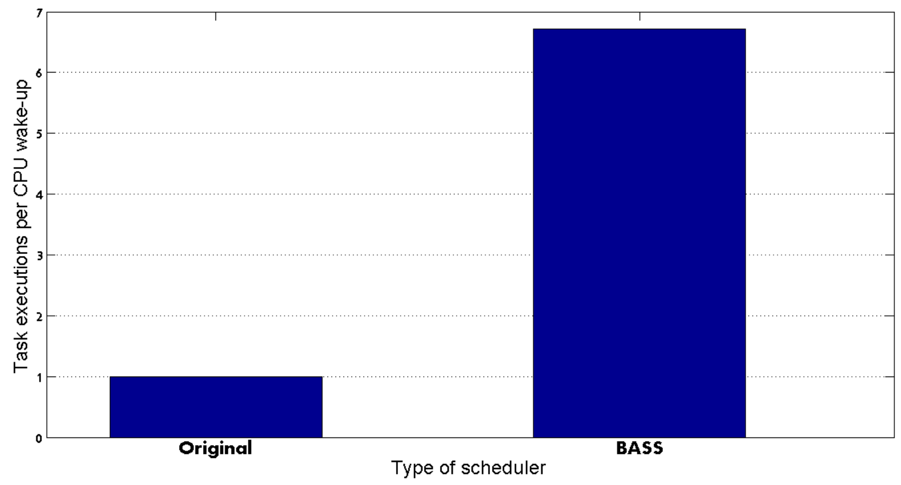
The first test case in our experimental procedure is the use of Rheuma App with the default time intervals. GCD method is totally applicable as the time intervals (60 and 300 seconds) have *gcd* equal to 60 (i.e. greater than 1). In particular, BASS generates and executes a cumulative task consisted of s_{state} to $s_{prox.}$ and s_{state} to $s_{loc.}$ every 60 and 300 seconds respectively. F-GCD and Opportunistic Execution are just as effective in this case as all the tasks are already merged due to GCD and no other application is running to cause extra CPU wake-ups. For this reason, flexibility is set to 0.

With regard to the original scheduling policy that is implemented, once a sensor sampling task is initiated it is periodically executed according to its time interval. In this case, the initiation time of each task differs by 6 seconds from the previous one so as to avoid accidental overlapping. We want to compare the energy efficiency before and after applying BASS on Reuma App, thus we measure the following:

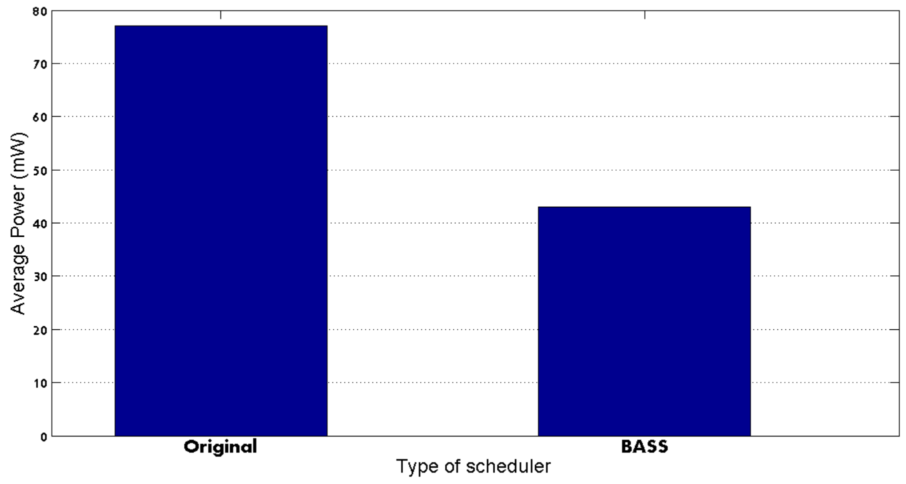
1. The number of task executions per CPU wake-up (see Figure 5.1a)

2. The average power (mW) required for the Rheuma App (see Figure 5.1b)

As shown in our measurements, it turns out that BASS, and particularly GCD, makes Rheuma App clearly more energy efficient than it is by default. Supporting this, the task executions per CPU wake-up are increased by 571%, while the average power required in HTC Sensation is reduced by 44% compared to the original scheduler. This gain is consistent with the measurements presented in Chapter 3 if we consider that the combination of all four tasks resulted in 40% and 45% energy savings in Samsung Galaxy Gio and Galaxy S II, respectively.



(a) Number of tasks executed per wake-up.



(b) Average Power (mW) required in HTC Sensation.

Figure 5.1: Energy efficiency before and after applying BASS on Reuma App.

5.1.2 Case 2: Worst Case Time Intervals

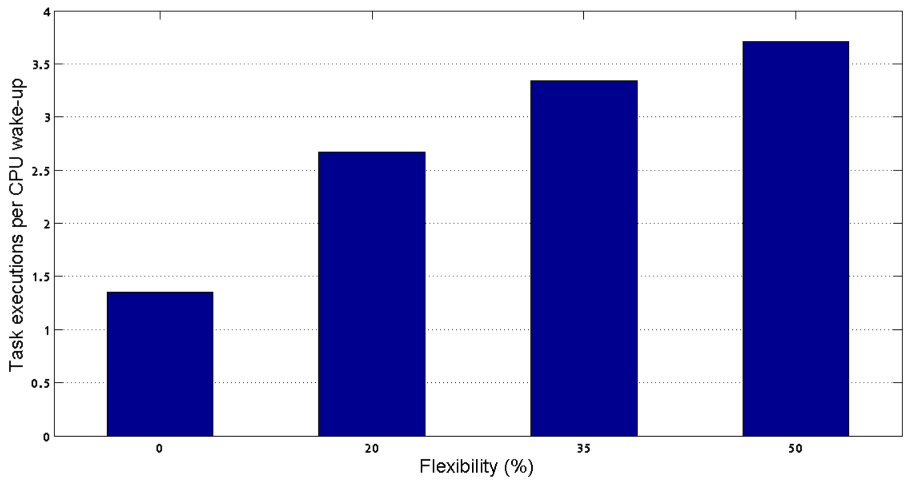
In our second test case we aim to evaluate the effectiveness of F-GCD. To this end, we reset the time intervals of the seven sensor sampling tasks. In order to test F-GCD we have to eliminate the effect of GCD, thus prime numbers (from 41 up to 79) are used as time intervals to guarantee that *gcd* is always equal to 1. We apply four different flexibilities (0%, 20%, 35% and 50% of the corresponding time interval) to observe the relation between the flexibility and energy efficiency. It has to be mentioned that for 0% flexibility the behaviour of Rheuma App is identical to the original scheduling because both F-GCD and GCD are ineffective. Similarly to the first case, the experiments conducted measure the number of task executions per CPU wake-up (see Figure 5.2a) and the average power (mW) required for the Rheuma App (see Figure 5.2b).

As it can be observed, the number of tasks executed for 0% is greater than 1. This is because two tasks are considered to overlap when the second starts its execution while the first still executes. Applying 20%, 35% and 50% flexibility increases the number of tasks executed per CPU wake-up by 98%, 147% and 174% respectively while it reduces the average power by 26%, 33% and 37%. As shown, 20% flexibility has quite an impact on energy efficiency of Reuma App. Raising the flexibility above 35% has minor effect.

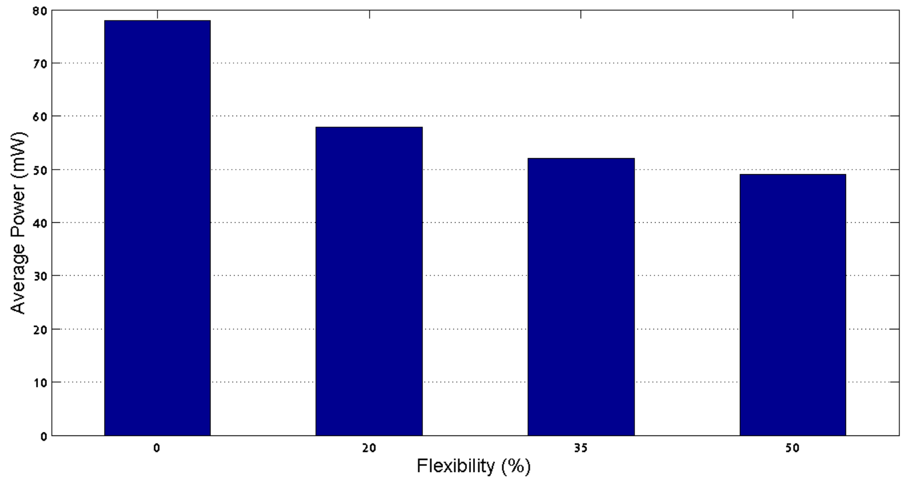
5.1.3 Case 3: Extra Application

As we explain in Chapter 4, the Opportunistic Execution method seeks to maximize the overlapping between the execution of the sensor sampling tasks and the CPU wake-ups due to other applications. In our last test case we exam the efficacy of Opportunistic Execution. For this purpose, we create an application that wakes up the CPU in order to obtain one accelerometer sample every minute. Every time interval for executing the task is uniform randomly chosen in $0 - 60$ seconds interval. We name this application Extra App and set it to run in parallel with Rheuma App. Hence, we initially measure the number of overlaps between the Rheuma App tasks and the Extra App for different flexibilities (see Figure 5.3a). To express as a percentage, we measure the overlaps occurred during 100 task executions of Rheuma App. Taking into account the average power required for Rheuma App, we also measure the overhead caused by the Extra App (see Figure 5.3b).

As shown in the figures, the effect of flexibility follows a pattern similar to the one in the second test case. The overlaps between the tasks of the two applications are increased by 31% when we apply 20% flexibility while they are slightly more (46%) in case of 50% flexibility. Similarly, the average power overhead due to the Extra App is reduced by 21% and 32% when we apply 20% and 50% flexibility respectively. In both the second and third

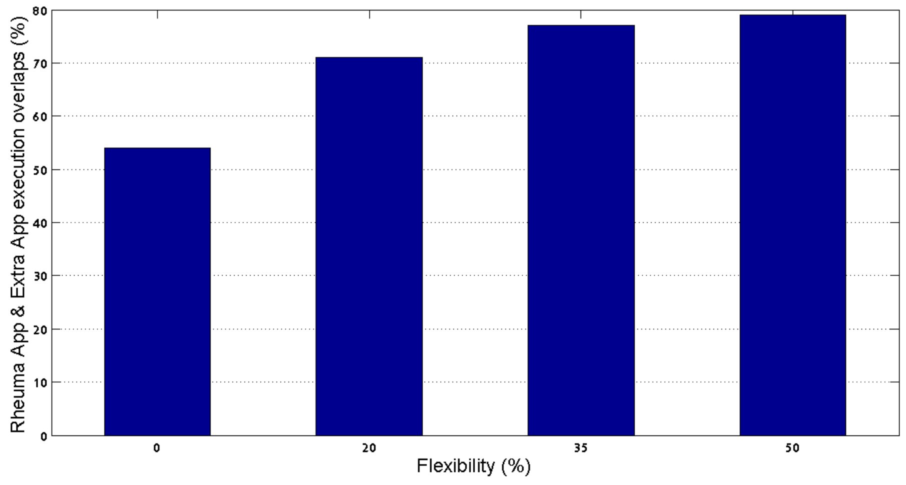


(a) Number of tasks executed per wake-up.

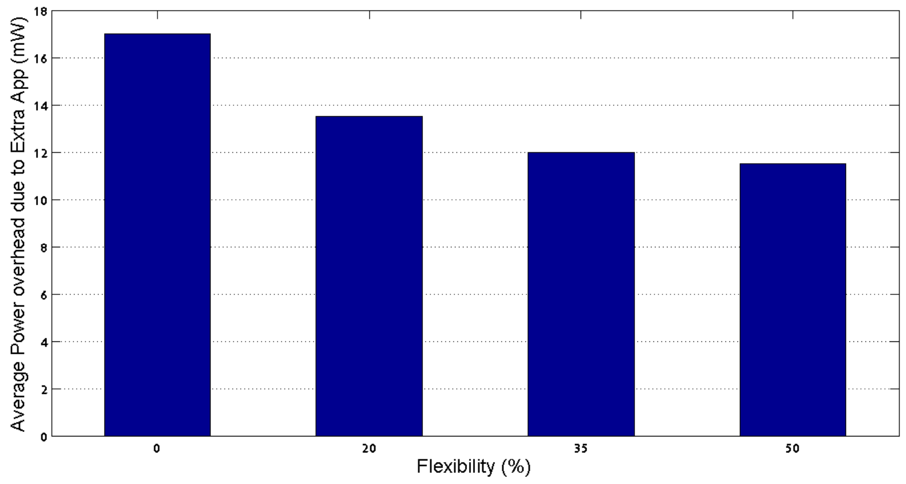


(b) Average Power (mW) required in HTC Sensation.

Figure 5.2: Energy efficiency when different flexibilities are applied.



(a) Number of overlaps between Rheuma App and Extra App.



(b) Average Power overhead (mW) in HTC Sensation due to Extra App.

Figure 5.3: Energy efficiency when Extra App is running and different flexibilities are applied.

test cases, increasing the flexibility above 35% is barely effective.

5.1.4 Real-World Experiment

In addition to the measurements analysed in the three test cases, a real-world experiment was also conducted. In this case, the parameters of the first test case (i.e. the default settings of Rheuma App) were used. Each experiment lasted one day and a user followed his regular daily pattern. For the duration of the experiment the airplane mode was off and the collected data was transmitted every 30 minutes. Hence power was consumed in transmission over 3G network and Wi-Fi. The results showed a 18% reduction in energy consumption. Note that the battery estimation provided by the phone is quite inaccurate (use of Monsoon was impossible due to lack of mobility) and it would be inexact to compare this percentage to the 44% gain measured with the Monsoon. In case of such a comparison, extensive measurements regarding the energy consumed in the data transmission should be conducted. However, this percentage can be considered as an indication that BASS conserves energy in case of smartphone regular usage.

5.2 Discussion

Throughout our experimental procedure several observations were made:

- The three batch scheduling methods that are implemented in BASS increase the energy efficiency in sensing. As it is observed in all of our test cases, BASS conserved energy by reducing the number of CPU wake-ups needed for the execution of the same number of tasks.
- Raising the flexibility from 0% to 20% has a major impact. On the contrary, as shown in the results, a flexibility greater than 35% is relatively pointless. However, the flexibility margin always depends on the application on which BASS is applied and the limit differs according to the sampling time accuracy that is required.

Moreover, two main obstacles were encountered during our evaluation process and have to be pointed out:

1. Initial experiments were conducted with a ZTE Blade smartphone. However, it turned out that the use of the proximity sensor was resulting in extensive CPU usage. The reason was that a wakelock was kept even when release function was called. The results were discarded.
2. During the observation of energy measurements there was no easy way to map the energy consumption samples depicted by Monsoon to the corresponding sensor sampling tasks. Thus, our experiments

were repeated several times in order to make sure that our setup was correct.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The main goal of this thesis was to provide an application and user agnostic energy saving method which eliminates the CPU wake-up overhead due to the periodic sensing on smartphones. BASS reduces the number of CPU wake-ups in such a way as to capture accurate sensor data (in terms of sampling time) without draining the device battery.

The main principle behind BASS is to schedule the execution of the sensor sampling tasks in batches so that fewer CPU wake-ups are needed. In particular, BASS detects when two or more tasks can be scheduled to overlap each other or with the execution of another application task. To this end, it applies three different batch scheduling methods: GCD, F-GCD and Opportunistic Execution.

From the experiments performed, we derived that BASS reduces the power consumption of our testbed application by up to 44% in laboratory environment and 18% in real-world. Furthermore, BASS achieves power savings of 37% in a worst case scenario while it is able to conserve 32% energy from another application that is running simultaneously with our testbed.

Our solution provides a robust framework. It can be applied to different applications that execute periodic sensing tasks. Furthermore, it guarantees energy conservation independently of the subset of sensors that are used. Introducing the parameter of flexibility, it exploits the different levels of sampling time accuracy demanded in order to provide energy savings.

In conclusion, even if some aspects of BASS might allow further improvement, it describes a compact approach for energy efficient sensing in smartphones, which completes the missing parts of past solutions. It is worth mentioning that Sense has already embedded BASS to various commercial sensing applications that are daily used by its customers.

6.2 Future Work

There are several ways to enhance BASS :

- Further experimentation should be conducted in order to identify a model that describes the relation between flexibility and energy efficiency. Extensive real-world experiments should also take place to ascertain if BASS conserves energy on a daily usage basis.

- Our flexibility parameter describes the allowable accuracy loss in terms of task time of execution. However, it is uncertain if this elasticity results in accuracy loss regarding state recognition. To this end, experimentation using several relevant applications (e.g. activity recognition application) should be conducted to investigate if the accuracy loss in terms of task time of execution implies accuracy loss in the state that is recognized.

- Regarding our energy analysis in Chapter 3, further investigation in various smartphones and sensors has to be performed. Depending on the results it may turn out that in different phones different sensor combinations are preferable and thus the scheduling policy has to be adjustable.

- Finally, our long-term goal is the development of a middle-ware tool based on the concept of BASS that performs batch scheduling, synchronizing the task execution between all of the applications running in a smartphone.

Bibliography

[1] Alarm manager in android. `http://developer.android.com/reference/android/app/AlarmManager.html`.

[2] Android logging system logcat. `http://developer.android.com/tools/help/logcat.html`.

[3] Android operating system. `http://www.android.com`.

[4] Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor`.

[5] Rheumatoid arthritis. `http://www.ra.com`.

[6] Sense android library. `https://github.com/senseobservationsystems/sense-android-library`.

[7] Sense observation systems. `http://www.sense-os.nl`.

[8] Sense platform. `https://play.google.com/store/apps/details?id=nl.sense_os.app`.

[9] Wakelock in android. `http://developer.android.com/reference/android/os/PowerManager.WakeLock.html`.

[10] Hrishikesh Amur, Ripal Nathuji, Karsten Schwan, Mrinmoy Ghosh, and Hsien-Hsin S. Lee. Idlepower: Application-aware management of processor idle states. In *In Proceedings of MMCS, in conjunction with HPDC08*, pages 1–8, 2008.

[11] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *IMC09, November 46, Chicago, Illinois, USA*, pages 280–293, 2009.

[12] Niels Brouwers and Koen Langendoen. Pogo, a middleware for mobile phone sensing. In *Middleware '12 Proceedings of the 13th International Middleware Conference*, pages 21–40, 2012.

[13] Matt Calder and Mahesh K. Marina. Batch scheduling of recurrent applications for energy savings on mobile phones. In *7th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks (SECON)*, pages 1–3, 2010.

[14] Hung-Ching Chang, A.R. Agrawal, and K.W. Cameron. Energy-aware computing for android platforms. In *2011 International Conference on Energy Aware Computing (ICEAC)*, pages 1–4, 2011.

[15] Ionut Constandache, Shravan Gaonkar, Matt Sayler, Romit Roy Choudhury, and Landon Cox. Enloc: Energy-efficient localization for mobile phones. In *Infocom 2009, IEEE*, pages 2716–2720, 2009.

[16] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings*, pages 222–231, 1993.

[17] Qian Diao and Justin Song. Prediction of cpu idle-busy activity pattern. In *IEEE 14th International Symposium on High Performance Computer Architecture, HPCA 2008*, pages 27–36, 2008.

[18] Shravan Gaonkar, Jack Li, Romit Roy Choudhury, Landon Cox, and Al Schmidt. Micro-blog: Sharing and querying content through mobile phones and social participation. In *MobiSys08, Colorado, USA*, pages 174–186, 2008.

[19] Dawud Gordon, Jrgen Czerny, Takashi Miyaki, and Michael Beigl. Energy-efficient activity recognition using prediction. In *ISWC 2012: The 16th IEEE International Symposium on Wearable Computers*, pages 29–36, 2012.

[20] Stuart Hayes. Controlling processor c-state usage in linux. In *Technical report, Dell Inc.*, pages 1–7, 2012.

[21] Jingcao Hu and Radu Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings (Volume:1)*, pages 234–239, 2004.

[22] Alexander W. Min, Ren Wang, James Tsai, Mesut A. Ergin, and Tsung-Yuan Charlie Tai. Improving energy efficiency for mobile platforms by exploiting low-power sleep states. In *CF12, May 1517, Cagliari, Italy*, pages 133–142, 2012.

[23] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA '08 Proceedings of the 35th Annual International*, pages 63–74, 2008.

[24] Jeongyeup Paek, Joongheon Kim, and Ramesh Govindan. Energy-efficient rate-adaptive gps-based positioning for smartphones. In *MobiSys10, June 1518, San Francisco, California, USA*, pages 299–314, 2010.

[25] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle-do nothing, efficiently... In *Proceedings of the Linux Symposium, Volume Two, Ontario, Canada*, pages 119–126, 2007.

[26] Chris N. Potts and Mikhail Y. Kovalyov. Scheduling with batching: A review. In *European Journal of Operational Research 120*, pages 228–249, 2000.

[27] Bodhi Priyantha, Dimitrios Lymberopoulos, and Jie Liu. Little rock: Enabling energy-efficient continuous sensing on mobile phones. In *Pervasive Computing, IEEE (Volume:10 , Issue: 2)*, pages 12–15, 2011.

[28] Lixin Tang and Hua Gong. The coordination of transportation and batching scheduling. In *Applied Mathematical Modelling 33*, pages 3854–3862, 2009.

[29] Yi Wang, Bhaskar Krishnamachari, and Murali Annavaram. Semi-markovian user state estimation and policy optimization for energy efficient mobile sensing. In *9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 533–541, 2012.

[30] Yi Wang, Bhaskar Krishnamachari, Qing Zhao, and Murali Annavaram. Markov-optimal sensing policy for user state estimation in mobile devices. In *IPSN10, April 1216, Stockholm, Sweden*, pages 268–278, 2010.

[31] Zhixian Yan, Vigneshwaran Subbaraju, Dipanjan Chakraborty, Archan Misra, and Karl Aberer. Energy-efficient continuous activity recognition on mobile phones: An activity-adaptive approach. In *Proceedings of the 16th International Symposium on Wearable Computers (ISWC), Newcastle, UK*, pages 17–24, 2012.

Chapter 7

Appendix A

There are two different options in setting an Alarm Manager:

ELAPSED_REALTIME_WAKEUP The CPU will wake up in order to perform the scheduled task.

ELAPSED_REALTIME The execution of the scheduled task is performed once CPU is woken up due to an event (i.e. user presses a button or another application task is executed).

As shown in the sample of code below, for every upcoming task execution two different alarms are set:

```
mgr.set(AlarmManager.ELAPSED_REALTIME, (nextExecution -
    flexibility), opportunisticExecution);
mgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, nextExecution,
    deterministicExecution;
```

In case *opportunisticExecution* is able to be performed (i.e. the CPU is woken up some time between ($nextExecution - flexibility$) and *nextExecution* time) the *deterministicExecution* is cancelled.