Improving Energy Efficiency of Personal Sensing Applications with Heterogeneous Multi-Processors

Moo-Ryong Ra*, **Bodhi Priyantha**†, **Aman Kansal**†, **Jie Liu**†
University of Southern California* Microsoft Research Redmond†

ABSTRACT
The availability of multiple sensors on mobile devices offers a significant new capability to enable rich user and context aware applications. Many of these applications run in the background to continuously sense user context. However, running these applications on mobile devices can impose a significant stress on the battery life, and the use of supplementary low-power processors has been proposed on mobile devices for continuous background activities. In this paper, we experimentally and analytically investigate the design considerations that arise in the efficient use of the low power processor and provide a thorough understanding of the problem space. We answer fundamental questions such as which segments of the application are most efficient to be hosted on the low power processor, and how to select an appropriate low power processor. We discuss our measurements, analysis, and results using multiple low power processors and existing phone platforms.

INTRODUCTION
The ubiquity, portability, and connectivity of mobile phones make them ideal platforms for continuous sensing applications that are used to derive user context for a variety of purposes. Several such applications have been proposed and prototyped using phone based sensors [14].

One of the key challenges for continuous mobile sensing applications is their energy use and the resultant battery life impact. Sensors, other than GPS, along with associated signal processing, are often not a huge energy concern when used in interactive foreground applications. For example, when playing games, the display (300-800mW), the network (600-1400mW), and the processor activities for graphics rendering dominate the total application energy consumption. On the other hand, for many mobile sensing applications, the sensing activity is performed in the background, *continuously*, when the other components such as the display are not in use. Such continuous operation has a high energy overhead on current mobile phone architectures. One of the main reasons is that the phone's application processor (AP) is designed to handle intense user interactions such as touch and content rendering in a responsive manner, and is not energy efficient for performing continuous sensing tasks.

To solve this problem, prior work has proposed the use of an additional low power processor (LP) to control the sensors in mobile device platforms [3, 28, 18]. New mobile device processors such as TI OMAP^{TM}5 Platform [32] already offer such an on-chip low power processor core in addition to the main AP core. The basic tenet is that the LP consumes very low power in active state and has negligible wakeup overheads. It can thus execute simple repetitive sensing tasks very efficiently, allowing the AP to stay in sleep mode unless a computationally intensive task is to be performed. The energy consumption of continuously running background sensing tasks on the LP is similar to the idle energy consumption of the phone [28] and hence acceptable. Programming abstractions that facilitate application development using both processors have been recently proposed [18].

This new multi-processor architecture presents an additional challenge for application design. Decisions must be made on what functionality should be put on the LP and how to partition an application among the two processors. In addition to controlling sensors, the LP can also handle application logic as long as it fits in the memory. However, for most complex computational tasks, using the AP is not only faster but often consumes lower total energy as well. The partitioning decision is hence non-trivial.

There are at least three design choices available here. A first possibility is that the determination of which application components run on the LP may be made at run time, based on the overall demand for resources across the multiple applications active at any time. Such approaches have been explored for offloading to the cloud [29]. A second possibility is to let the developer determine the partition of the application at design time and compile the application appropriately for the two processors. Programming infrastructure for this approach has been developed in [18], assuming that the developer has determined the correct partition. A third possibility is to hardwire a library of methods on the LP that are expected to be useful for all applications. Developers write their applications only on the AP, utilizing the LP library for methods already provided.

In this paper, we investigate the partitioning of applications across the two processors from an energy efficiency perspective. We provide a methodology to analyze the application components to determine the most efficient placement. We quantify the impact of different active mode power consumptions and transition overheads across the two processors on the partitioning decision.

Specifically, we make the following contributions. First, we provide a methodology to analyze if a computational task is more efficient on the LP or the AP, and perform measurements to apply the analysis on a realistic set of computational tasks used in sensing applications. Second, we extend the analysis and measurements to study the energy efficiency effects of processor selection for complete applications that consist of multiple computational tasks. Third, we study the energy efficiency trade-offs for multiple applications executed simultaneously. Finally, based on the single task, complete application, and multi-application measurements, we derive guidelines to facilitate the decision of which processor a task should be placed on.

The above study allows us to quantitatively conclude which of the design choices in the use of multiple heterogeneous processors is the most efficient and effective. Our investigation suggests that the third approach of pre-determining a library of methods for the LP is the most energy efficient one in practice, and also offers additional advantages in terms of ease of use for developers. We also propose how limited amount of processor selection may be made at run time for additional energy gains, but the magnitude of the gains does not justify the complexity of supporting dynamic placements for app components. The second approach of partitioning each application at design time suffers from potentially incorrect and inefficient placements when multiple applications are combined.

Using a survey of sensing applications, we propose an example library of methods that are efficient to be provided on the LP for low energy operation of a wide variety of application classes. While our selection of methods for the library is motivated based on the applications currently envisioned, the methodology developed allows updating this selection if new types of applications are envisioned.

RELATED WORK

Recently proposed Reflex [18] platform provides easier programming experience in multi-processor mobile platforms. They wrap complex low-level development issues in LPs using distributed shared memory abstraction, thereby the programmer can use standard development techniques to build applications. LittleRock [28] realizes a proof-of-concept prototype to emphasize the effectiveness of the low power processor, also inspires our work. Unlike these works, we provide a partitioning guideline as well as proper runtime design principles to enable continuous sensing applications. Our work is complementary to both works since our guidelines can be realized on top of those platforms.

There are several works on workload partitioning across heterogeneous platforms. MAUI [7] and Goraczko et al. [10] proposed integer linear programming based partitioning approach. The former is across mobile devices and the cloud. The latter is between two processors. Unlike these works, our work uses simulation-based approach to derive more accurate results closer to reality. Because wakeup and sleep transitions depend on actual job scheduling at runtime, adjusting ILP-based optimization to our setup is not an easy task. There are other partitioning approaches as follows. Wishbone [25] uses profile-based compile-time optimization method to determine partitions under the context of sensor network. Odessa [29] dynamically offloads compute stage at runtime between mobile and cloud platform based on runtime profiles. Chroma [1] switches its configuration at runtime among pre-defined set of partitions either programmer-specified or suggested by domain expert. These works have either significantly different goals, e.g. fps and makespan in [29], different partitioining guidelines in [25, 1].

Our work is inspired by emerging demands of continuous sensing applications. Numerous continuous sensing applications have been developed in research community. Part of them are illustrated in Table 3. More in-depth exploration can be found in [14].

With the emergence of continuous sensing applications, the needs on systems support for those applications naturally arise. A set of prior work [31, 27, 21] have proposed filtering out uninteresting data using simpler processing states in the inference pipeline. Despite of limited similarity, our investigation explores the generic APIs that will help app developers to exploit such pre-filtering in a heterogeneous multi-processor architecture for energy efficiency. MyExperience [9] proposes a framework that can combine both objective, e.g. sensor readings/classification results, and subjective sensing activities. SeeMon [11] suggests context monitoring framework for mobile devices with many sensors. Focusing on the energy consumption, the framework provides a way to select a relevant set of sensors, intuitive programming abstraction to the developers. Jigsaw [21] identifies common components for sensing tasks per sensor, optimizes related components so that multiple applications can adaptively and effectively utilize the sensor functionality. The Mobile Sensing Platform [5] equips various sensors, and provides a suit of feature extraction and classification methods in a dedicated hardware.

ENERGY PROPORTIONALITY

The goal of defining energy proportionality is to quantify the energy "wasted" in addition to executing the application logic and to compare the energy efficiency across different application configurations. An application configuration refers to a set of executing applications with some portion of the processing chain of each application executing on the LP. Given a platform P and a set of applications, we first introduce the notion of *ideal energy consumption* (IEC) of executing these applications on P. We define an ideal platform $\hat{P}$ as a platform with the same performance and energy characteristics of P, except $\hat{P}$ has zero sleep power consumption and takes zero time and energy to transition from sleep state to active state. The ideal energy consumption (IEC) of a set of applications on P, $\hat{E}^P$ is the energy consumed when executing the applications on the AP of $\hat{P}$. In reality, all the

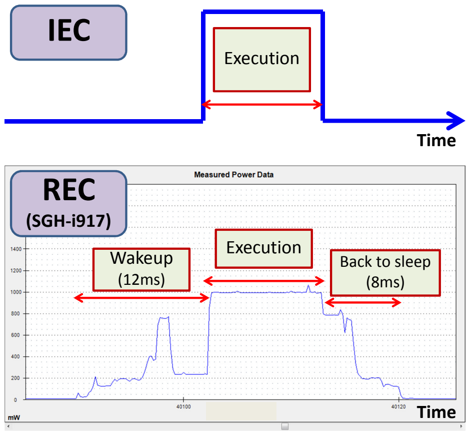


Figure 1. Running a piece of code on the main processor(AP)

processors have some non-zero sleep power consumption, and have to pay a transition cost when transitioning between sleep and wake up. We denote the real energy consumption (REC) of the application on P by E^P.

We illustrate the difference between REC and IEC with a concrete measurement from a mass market mobile phone – Samsung Focus SGH-i917. Figure 1 shows the energy trace collected when the phone woke up from the sleep mode, executed a user application that stresses the CPU to 100%, and went back to sleep. Ideally, if all energy is used by executing the user application, the profile of power consumption will have the same shape as the execution. However, due to the process of waking up the phone and then putting it back to sleep, the real energy consumption is significantly higher.

On an ideal platform, the system can enter the sleep state whenever there is a gap in the execution of applications. In real platforms, however, whether the platform can go to sleep depends on whether the gaps between executions are greater than the time it takes to make state transitions, known as the *break-even time* [2, 8]. For example, if the time that an application waits for its inputs is less than the break-even time, the whole platform has to stay in the active mode even though there is nothing to execute, which we call the *idle* mode.

To qualify the difference between REC and IEC, we define the *Energy Proportionality Factor* (EPF) of running an application configuration A on platform P as

$$EPF_A^P = \frac{E_A^P}{\hat{E}_A^P}.$$

When the platform is obvious, we also write EPF_A for short. If all applications are run on the AP, since no real platform has zero transition energy, EPF will be greater than 1. However, in application configurations where processing is shared between AP and LP, EPF can be smaller than 1 if application stages run more efficiently on LP to offset the transition costs. Given a platform P, our aim is to determine the application configuration with the smallest EPF.

OPTIMIZING EPF ON TWO-PROCESSORS

Next we examine how to use the notion of energy proportionality to determine energy efficient application configurations that span AP and LP. Here we assume a single application being executed. During evaluation, we extend this to multiple concurrent applications. We assume that an application consists of several computational stages wired together either sequentially or conditionally. Certain stages may not be executed for all inputs, such as certain complex signal analysis may not be performed when the sampling and thresholding reveals that the signal magnitude is close to the noise floor. The specific computations that comprise mobile sensing applications are described in the next section.

Assumptions To make the analysis tractable, we begin with a few assumptions some of which are later removed. First, we assume that a sensing application starts with sensor data sampling, and its execution is periodic. Second, we assume that the execution time of a particular compute stage, for the input instances where the stage is executed, is constant and does not depend significantly on the input. In reality, certain inputs such as all zeros, may simplify the computation leading to faster execution of a given stage but for now we assume such differences are insignificant. We also assume that communication cost between the two processors is negligible, compared to the execution time and the sampling period. This is a realistic assumption since most new platforms have the AP and LP on the same chip; even if they are implemented as two chips, they would be connected by a fast bus.

Fourth, since we specifically use a low power microcontrollers (LP) to offload sensing, we assume (which is confirmed by measurements in next section) that the sleep to active and active to sleep state transition costs for the LP, as well as the idle power consumption of the LP in sleep state, are all negligible relative to other energy parameters involved. We also assume that all computation stages are *schedulable* regardless of which processor they are placed on. This means that the sum of the execution times of all the compute stages of the application is lower than the sampling rate governed periodicity of the application execution. However, we note that, particularly when multiple applications are executing simultaneously, the LP may not be fast enough to keep all the tasks schedulable. We remove this assumption later in the evaluation section.

Total Energy Consumption Suppose that one invocation of an application consists of N stages, with possible repetitions of certain stages, and lasts one period of duration d. We use the notations in Table 1 in the analysis.

On the application processor, M, the ideal energy consumption(IEC) of the application is

$$\hat{E}^M = \sum_{i=1}^{N} \left[F_i \cdot P_{active}^M \cdot T_i^M \right] \tag{1}$$

In reality, the transition energy and sleep energy in applica-

Variable	Description
$E_i^{\{cpu\}}(\hat{E}_i^{\{cpu\}})$	(Ideal) energy consumption contributed by the execution of a stage i. cpu could be M for AP, L for LP.
d	The duration of the execution.
$P_{active}^{\{cpu\}}$	Power consumption when the cpu is in active state. We assume same idle and active power consumption.
$P_{sleep}^{\{cpu\}}$	Power consumption when the cpu is in sleep state.
$T_i^{\{cpu\}}$	Execution time of stage i on the $\{cpu\}$.
E^{trans}	Transition energy cost for AP.
K	# of mode transitions on AP in a given duration d, since we ignore transitions on LP.
N	# of compute-stages.
s_i	Slow-down factor of the stage i. Formally $s_i = \frac{T_i^L}{T_i^M}$
F_i	Expected number of times that stage i is executed in a given duration d.
L_i	Placement variable of the stage i, which takes either 0 or 1. If 1, the stage i runs on the main processor, otherwise it runs on the low-power processor.

Table 1. Variables used in the modeling effort

tion processor cannot be ignored. So, real energy consumption(REC) (on the application processor) is:

$$E^M = \hat{E}^M + E^{trans} \cdot K + P_{sleep}^M \left(d - \sum_{i=1}^{N} (F_i\, T_i^M)\right) \quad (2)$$

where $E^{trans} \cdot K$ is the energy spent on mode transitions. Here we ignore the time spent transitioning between the two states since d (in the order of 10's of minutes or hours) is much larger than the total transition time (in the order of seconds).

When there are two processors, M and L, we use a placement variable L_i to determine the energy, L_i takes the value 1 when a stage is placed on the AP (M), or 0 if on the LP (L). The active energy use can be broken down into its constituent portions of active and transition energy and be written as:

$$\begin{aligned} E^{multi} = & \sum_{i=1}^{N} [L_i \left(F_i (P_{active}^M - P_{sleep}^M) T_i^M\right) \\ & + (1 - L_i)(F_i P_{active}^L T_i^L)] + E^{trans} K' + P_{sleep}^M d \end{aligned} \quad (3)$$

Notice that, since we assigned some stages to LP, the number of transitions in the AP, K' is different from the K in Eqn. (2), and it can be different from the sum of F_i since adjacent execution stages may get coalesced or dispersed depending on the placement and the specifics of task scheduling. Comparing E^{multi} and E^M, it is obvious that the potential gain at energy proportionality comes from the trade off between the reduction of the number of transitions and the energy inefficiency of LP.

Computation Stage Placement To determine the partition that results in best EPF, we use some additional characteristics of typical mobile continuous sensing applications. For these applications, typically, along the compute pipeline, the output rate of each stage tends to significantly decrease. For instance, sampling and buffering stage may execute at 100Hz to collect and buffer sensor samples, however, activity classification that operates over a buffer of data executes at a much slower rate, e.g. at 1Hz.

On the other hand, as we observe in the next section, stages at the start of the pipeline tend to be more light weight, performing simple buffering and filtering of data, while stages further into the pipeline tend to be more compute intensive.

Given the smaller transition cost of LP, and that computationally heavy tasks run more efficiently on AP (computational efficiency more than makes up for the wake up cost), it is very likely that the optimum partition would be a simple cut of the processing chain. Hence, we partition an application by cutting the processing chain into two; where the fist several stages are assigned to a LP while rest of the chain is assigned to AP. Specifically, when partitioning an application, we start from the very first processing stage and incrementally improve EPF by examining the placement of consecutive compute stages. We stop when we reach the first stage that does not improve EPF when assigned to LP (later we add additional constraints such as resource restrictions when assigning tasks to LP). Our measurement results verify the assumption of a simple cut of the processing chain.

When deciding the placement of a specific stage i, under our scheduleability assumption, the only deciding factor is the relative EPF(energy) difference between the two processors. We assign a stage i to the LP only if the corresponding EPF reduces compared to assigning it to AP.

This relative EPF difference relying on the placement of stage i is denoted as Δ^i, and can be calculated as following. Let current app partition between two processor be $D[i \rightarrow M]$ when stage i is on AP, $D[i \rightarrow L]$ when it is on LP.

$$\begin{aligned} \Delta_i &= EPF^{D[i \rightarrow M]} - EPF^{D[i \rightarrow L]} \\ &= \frac{E_i^M - E_i^L}{\sum_{i=1}^N \hat{E}_i^M} \\ &= \frac{F_i\, T_i^M \left[P_{active}^M - P_{sleep}^M - s_i * P_{active}^L\right] \;+\; E^{trans}\, K'}{\sum_{i=1}^N \hat{E}_i^M} \end{aligned} \quad (4)$$

Note that new variable K' denotes the difference in number of transitions due to the different stage placement. Using Eqn. (4), one can compute where Δ_i is greater than 0 and then place the computation on the LP (setting $L_i = 0$). The equation clearly shows the key parameters that affect the placement of the computation stage. One of the factors that appears is s_i, which is the ratio of the execution time on the LP to that on the AP. This can be thought of as the slow-down factor for the LP compared to the AP. Given a particular hardware configuration for the AP and the LP, most of the

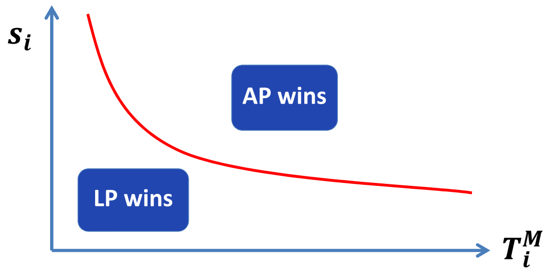


Figure 2. Partitioning Decision: Break-Even Graph

parameters in the equation are predetermined. What varies for the specific computation stage is s_i. Re-writing the condition for the case when LP wins, that is, $\Delta_i > 0$ assuming the worst case $K' \approx F_i$ to bring out s_i, we get:

$$\Delta^i > 0$$

$$\Rightarrow F_i\, T_i^M \left[P^M_{active} - P^M_{sleep} - s_i * P^L_{active}\right] + E^{trans} K' > 0$$

$$\approx s_i < \frac{P^M_{active} - P^M_{sleep}}{P^L_{active}} + \frac{[E^{trans}/P^L_{active}]}{T_i^M} \qquad (5)$$

Eqn. (5) can be used to draw a break-even line for the value of s_i below which using the LP is advantageous, as shown in Figure 2. The actual s_i is difficult to assess analytically since it can depend on several factors including the availability of data parallel instructions such as SIMD, floating point unit, memory size of the processor, bus speed, DMA availability, cache size, and the processor frequencies. It is best measured experimentally, and we perform such measurements in the next section to explore the placement decisions for realistic mobile sensing applications.

MEASUREMENTS ON HW/SW COMPONENTS

The goal of the measurement study is twofold: (1) getting experimental values for hardware-dependent parameters including the active and transition energy costs, and (2) measuring the slow-down factor s_i to explore the break-even graph (Figure 2) for concrete examples and extract useful insights on execution time difference between two processors. The hardware parameters can be measured directly and plugged into the anaylsis result in the previous section. To measure the software dependent slow-down and break-even parameters, we take common computational components from a survey of several mobile sensing applications, implement or port them to both platforms, and measure their execution times. We discuss both of these measurements in this section after describing the measurement platform.

Hardware Parameter Measurements

At the time of this study, there is no common smartphone platform available with a LP in addition to the main processor(AP). Hence for measurements related to the AP, we use a recent smartphone, Samsung SGH-i917, running the Windows Phone 7.5 operating system. The smartphone is equipped with a Qualcomm Snapdragon QSD8250 RISC processor. For the LP, we use three representative microcontrollers with varying HW capability: Atmel ATmega1284P (AVR), TI MSP430F5438 (MSP), and ST Micro ARM CortexM4 STM32F407VGT6 (ARM). Both the AVR and the MSP have a HW multiplier. The ARM processor has a dedicated single precision floating point unit and supports DSP instructions. For measuring the smartphone power consumption, we used the Monsoon Power Monitor. For LP power and execution timing measurements, we used a Textronix TDS3054B oscilloscope. When impossible to measure, the LP transition energy and time values are derived from the worst case approximation based on respective datasheets.

	AP	AVR	MSP430	ARM
CPU frequency (MHz)	1000	12	18	168
CPU bus width (bits)	32	8	16	32
Sleep state power (uW)	9200	1.8	6	2000
Active state power (mW)	1000	21	15	152
Wake-up delay(us)	12000	1	5	110
Sleep delay (us)	8000	(NA)	5	(NA)
Wakeup energy (uJ)	3101	0.006	0.03	0.66
Sleep energy (uJ)	3065	(NA)	0.075	(NA)

Table 2. Processor Parameters (The stop mode was selected as the sleep mode for the ARM. AVR & MSP - wakeup using internal clocks).

The hardware dependent parameters measured are reported in Table 2. The numbers for the AP are measured directly, and include the effect of any transition overheads coming from system components other than the AP. The LP power numbers reflect only the power consumed by the LP. We assume that the communication cost between the AP and the LP will be negligible when they are either part of the same on-board chipset or even part of a single System-on-Chip (SoC). It is easy to see from the table that the transition energy cost for the AP ($\approx$3 mJ) is several orders of magnitude more than that of the LP and the assumption regarding ignoring the LP transition costs is thus justified. Second, the power consumption of the AP in sleep state is quite significant and about the same order of magnitude as the active power for the LP.

Measuring Application Characteristics

As discussed before, the slow-down factor is highly computation dependent. To ensure that our measurements and the resultant conclusions are generally applicable, it is important to measure the energy use for computational tasks performed by representative applications. Since the range of possible applications is very large, to make this study tractable, we focus specifically on the parts of the application related to continuous sensing and corresponding inference that are likely to be executed in a continuous, background mode. In order to identify such common computational tasks, we begin with a survey of sensing applications published in the literature. For the rest of this section, we use MSP430F5438 as the representative LP.

Common Structure of the Continuous Sensing Applications

Table 3 summarizes the list of applications surveyed. Most sensing applications process the sensor data to detect or infer

Application	Description
UbiFit System	Devises a glanceable display to encourage physical activity based on sensed and inferred user status. [6]
Lester et al.	Esitmates daily caloric expenditure based on the sensor reading to help people's weight control. [17]
Lester et al.	Identifies physical activities (sitting, standing, walking, riding) for personal fitness, elder care, etc. [16]
Playful Bottle	Uses camera and accelerometer to estimate how much water a user drinks. [4]
KidCam	Records children's daily activities by designing a camera-based mobile sensing device. [12]
SensLoc	Identifies semantically meaningful places. Activates only a necessary set of sensors to optimize energy consumption. [13]
Lifelogging	Records daily life events for the people who have episodic memory impairement. Uses camera, audio, and GPS to log everyday life. [15]
SoundSense	Uses microphone to recognize human voice as well as categorize the ambient sounds using audio signal processing techniques. [20]
SpeakerSense	Recognizes who the speaker is. Partitions the computation pipeline across two different processors in order to conserve energy. [19]
Maekawa et al.	Recognizes daily activities using dedicated sensors (wrist mounted sensors and image processing components). [22]
Darwin Phone	Collaborative machine learning. Training data is shared among a multiple phones to augment the classification model. [23]
Nericell	Monitors road bumps and traffic using accelero-menter, microphone, and GPS sensors. [24]
PEIR	Analyzes GPS traces from the smartphones to provide a personal carbon exposure report. [**?**]
HealthGear	Uses a blood oximeter sensor to monitor user's physiological signals. [26]
EmotionSense	Combine several sensor readings from accelerometer, audio, GPS, BlueTooth to recognize people's emotional status. [30]
Fall Detector	Detects falls, crucial for the elderly. [33]

Table 3. Sensing applications surveyed

App. & Sensors	Sampling & Buffering	Feature Extraction	Classification
Intel MSP - Various	2∼550 Hz	0.1∼10 Hz	$<<$ 0.1 Hz
SoundSense - Audio	4, 8, 16 kHz	16 or 1.56 Hz	$<$ 1 Hz
SenseLoc - Accel.	4∼50 Hz	0.1 Hz	0.1 Hz
SenseLoc - GPS	1/60∼1/10 Hz	$<$ 1 Hz	$<$ 1 Hz
SenseLoc - WiFi	1/30∼1/10 Hz	1/60∼1/30 Hz	1/60∼1/30 Hz
PEIR - GPS	1/30 Hz	$<<$ 1 Hz	$<<$ 1 Hz

Table 4. The Stage Invocation Rate

Components	Description
Integer Operations	$+, \times, \div$ with different iteration counts
Floating Point Operations	$+, \times, \div$ with different iteration counts
FFT	input size 8∼256 [1]
MFCC	input size 8∼1024
Activity Recognition	Custom Implementation

Table 5. Measured Software Components

certain attributes of interest. The key observation from the survey is that most applications are comprised of a series of compute-stages, as follows.

Sampling and Buffering: Every sensing application must sample the sensors of interest, either periodically or using a custom sampling strategy, and put them into a data buffer. Application dependent windowing and framing may be performed. Some applications [19, 20] require overlapped framing to ensure the quality of classification.

Filtering (optional): Some applications may filter out uninteresting sampled signals, e.g. SoundSense [20], an application that needs sound data, may filter out all data samples corresponding to silence. The filtering stage may determine if the subsequent (possibly energy expensive) stages are executed at all. Since the filtering stage is executed almost always, it often uses computationally light-weight features such as zero-crossing rates, pitch detection, etc.

Feature Extraction: This stage extracts necessary features for the classification stage. The features computed depend on the type of the application. For instance, an activity recognition application [5] using a Bayes' classifier may use raw sensor samples. Others such as [19, 20] compute more complex features like Mel-Frequency Cepstral Coefficients (MFCC). Machine learning based applications may compute multiple simple features simultaneously.

Classification: This stage entails performing the application specific classification. Typically probability-based or machine learning based algorithms are used, which often involve heavy floating point operations and iterative computations. Among all compute-stages involved, the classification stages are most computationally demanding.

Post Processing: Once the event is classified, it may trigger another continuous sensing compute-chain or other application specific task such as displaying user notifications or initiating network communications. Some work [4, 6] encourage users to do certain physical activities. Others [9, 13] invoke other compute processes, etc. We treat the post processing task as a separate application in this paper.

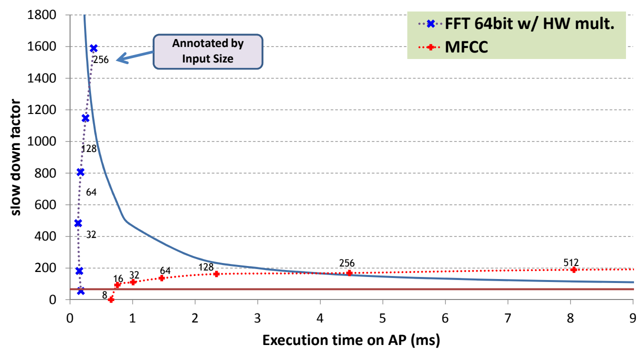
Frequency of Stage Execution

Table 4 illustrates some example rates of stage execution based on the surveyed work. Overall output rates of stage executions are dramatically decreased after we pass the sampling stage, and then remain as a small value. This is because many continuous sensing applications are highly related with human behaviors and human-perceivable rate is not as intense as individual sensor's sampling rates. As we can identify in the table, for the sensors such as microphone and accelerometer, the impact of frequent invocation can severely impact energy consumption, so we need to carefully adjust the observation when we design a related system.

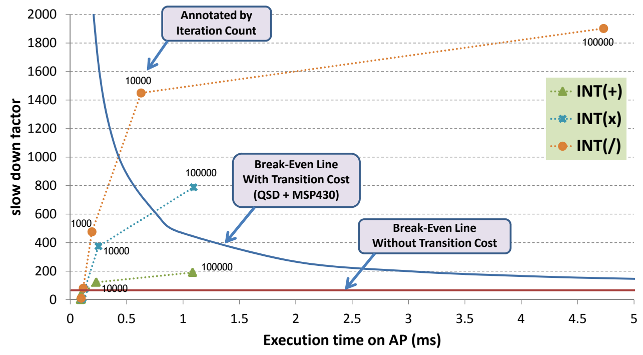
Slow-down Factor and Break Even Analysis

Table 5 summarizes the components that we measure on two processors. The two most prominent libraries in our survey were the Fast Fourier Transform and the Mel-Frequency Cepstral Coefficients. For a fair comparison, we used more storage intensive benchFFT on AP to take advantage its abundant storage while we used less storage intensive Kiss FFT [2]

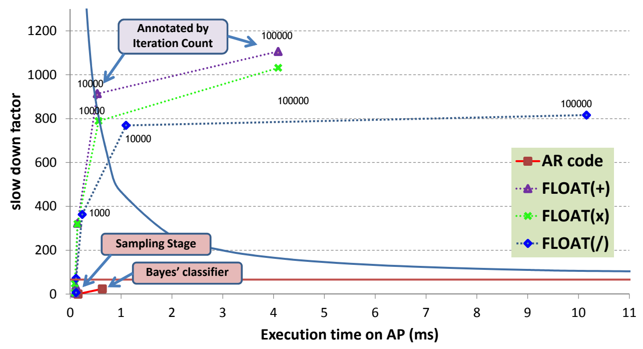
[2]http://sourceforge.net/projects/kissfft/



(a) Advanced Libraries



(b) Integer Operations



(c) Floating Point Operations

Figure 3. Measured results using common components in continuous sensing applications.

to profile the LPs. We used our own implementation for the MFCC library. In our measurement, we observe a lower bound of the unit operations on the AP, which is approximately 0.09 ms. We attribute this to various operating system resource management related overhead enforced by .NET framework on the phone.

The slow down factor measurement is plotted against the break-even curve based on Eqn. (5) in Figure 3. In this graph, we plot the actual break even curve based on the measured transition energy as well as an ideal break even curve (horizontal line) assuming zero transition energy. Based on these graphs, we identify three types of computing task (for routines such as FFT, for this discussion, we define a task by the combination of the computation and the particular input size), as follows. First, since we assume the worst case transitions, the tasks that lie above the break-even curve are always most energy efficient when executed on the AP, irrespective of how they are scheduled. Second, the tasks that lie below the ideal break-even curve are most energy efficient when executed on the LP, irrespective of the AP transition energy. Third, for the tasks that lie between these two curves, the most energy efficient processor assignment may depend on how they are scheduled. For example, if each of these tasks are schedules in an infinite back-to-back sequence, they will be more energy efficient when assigned to the AP since the impact of transition cost becomes negligible; however, if they are always scheduled individually, LP becomes the most efficient.

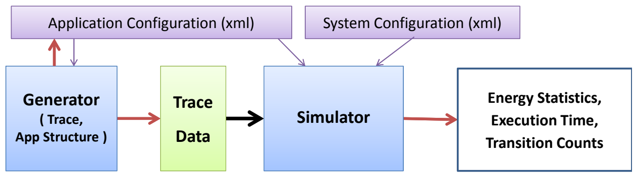

Application Configuration (xml)
System Configuration (xml)
Generator (Trace, App Structure)
Trace Data
Simulator
Energy Statistics, Execution Time, Transition Counts

Figure 4. Overview of the Simulator

Together with these observations, the frequency of execution may make a significant influence on overall energy consumption. Hence, to further explore real impact of the assignment policy, we developed a simulator that enables us to investigate various parameter combinations, thereby gives us a good intuition to determine the right assignment policy. The results are illustrated in the evaluation section.

EVALUATION

We quantify energy consumption of the system under realistic scenarios, using the following simulations and measurements.

Simulator. When we consider multiple application running simultaneously, the timing behavior of the computational tasks from all applications as well as the resultant numbers of transitions and sleep durations become more involved to be captured analytically. We built a simulator that operates at a fine time granularity to capture these effects. Figure 4 describes the architecture of the simulator. We use an event-driven simulator design. It takes three input files. The first input contains system configuration parameters, such as processor power consumption in every power state for both processors, CPU operating frequencies, scheduling policy, and transition energy cost. The second input describes the application structure in terms of compute stages and their wiring. The last input is a trace file of potentially multiple applications. The simulator computes the entire execution trace and outputs the desired statistics including total time spent on active and sleep state, average transition energy cost per processor, energy cost consumed by active and sleep state, the number of transitions occurred during execution, system-wide average power consumption, expected battery life assuming that the applications run continuously.

	QSD8250	MSP430	Slowdown Factor	Frequency of Execution
Sampling	0.15 ms	0.0365 ms	0.24	1-30 Hz
Classification	0.63 ms	15.02 ms	23.84	0.1 Hz

Table 6. Activity Recognition

	QSD8250	MSP430	Slowdown Factor	Frequency of Execution
Sampling	0.15 ms	0.0365 ms	0.24	1-15 Hz
FFT	0.38 ms	146 ms	384.21	1 Hz
MFCC	4.468 ms	753.71 ms	168.69	0.1 Hz
Classification	0.63 ms	15.02 ms	23.84	0.1 Hz

Table 7. Simplified SoundSense Application

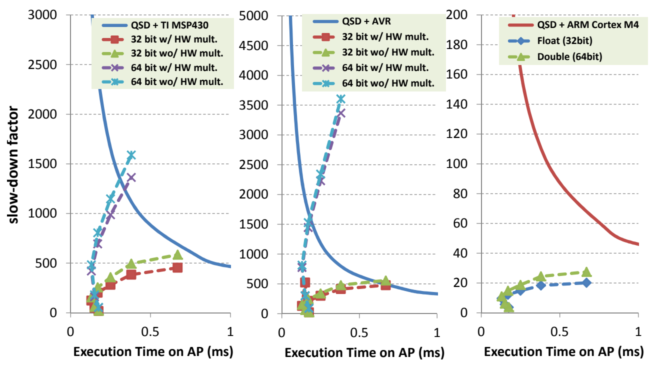


Figure 6. Break-Even Graphs using KissFFT for Different LPs

Computation Stage Placement

Using the simulator and measurement results we try to achieve three goals. First we do a first-order verification of our energy tradeoff analysis using two representative continuous sensing applications [16, 20] under reasonable simplifications. Next, we use the simulation framework to understand how concurrent applications impact the overall EPF (hence the total energy consumption). Finally, we use these results to derive a guideline for partitioning applications between AP and LP.

Applications. We choose two representative applications. The first application is an activity recognition application that consists of two compute-stages. The sampling stage takes accelerometer readings and stores them in a buffer. Then the classification stage uses the Naive Bayesian classifier to identify an activity based on the cumulative data in the buffer. The second application is a sound classification application, inspired by SoundSense [20]. We slightly simplify the application so that our measurement results can be used for profiling while preserving the reasonable application structure. This application is comprised of four stages; sampling, FFT, MFCC, and Bayesian classifier. To setup the simulator, we take measured data from Figure 3 to get execution times and slow-down factors for each compute-stage. We vary or set execution frequencies of individual components based on the original configuration. Table 6 and Table 7 contain details of these two applications.

Runtime Analysis on Applications. We first run each application alone. To evaluate the impact of partitioning decisions, we plot the EPF of the application execution against different partitioning strategies. For the partition strategy, we start with all the application components assigned to the AP; next we move one component at a time, starting form the sensor sampling component, to the LP and evaluate the EPF using the simulator. We do not evaluate all possible partitioning strategies due to two reasons; one is since a complicated partitioning is likely to incur heavy management overhead, the other is that the frequency of execution is likely to decrease as we go deep into the compute-pipeline.

Figure 5(a) shows the results from the activity recognition application. From this results we observe that almost all the reduction in EPF (hence, the energy) comes from the assignment of the high frequency sampling and buffering stage on to the LP. Beyond that, although the Naive Bayesian classifier lies well below the break-even line (Figure 3(c)), the incremental benefit from assigning it on the LP is small since the frequency of invocations of this stage is much smaller than the sampling stage.

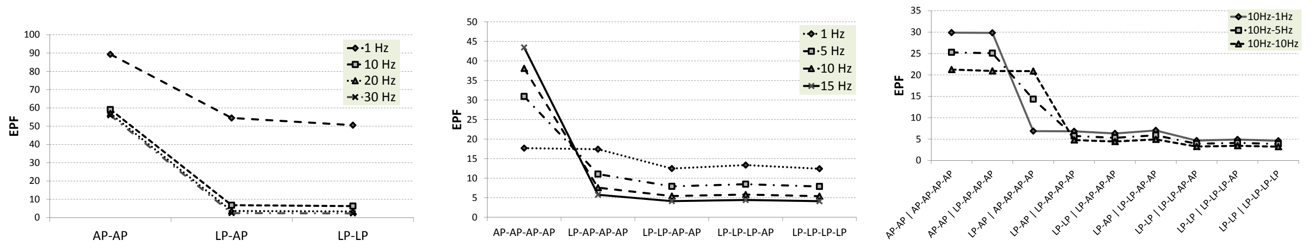
The case of more than two compute-stages running together are in Figure 5(b) and 5(c). We presented two such cases. Simplified sound sensing application has four stages and two application running together have 6 stages in total. Despite the interactions among those compute-stages, the results are qualitatively similar to the simple two stage case.

To more concretely relate the above results to actual battery lifetime on current phone hardware, consider the following hardware characteristics. Suppose that we have a 1500mAh battery, the one used in Samsung Focus i917 smartphone, fully charged at the beginning. We use the activity recognition app with 20 Hz sampling rate, continuously run it all the time until the entire battery is consumed. No other processes are running together. We use our simulator to execute the setting, and get the following results. LP-LP has 9.24 mW of average power consumption, 600.7 hours of battery life. LP-AP shows similar results; 9.96 mW and 557.33 hours. However, AP-AP consumes much more power, 147.79 mW, and ends up with only 37.5 hours of battery life. Unsurprisingly, the EPF difference in Figure 5(a) is directly connected to battery life time.

PLATFORM DESIGN IMPLICATIONS

In this section, we draw three guidelines for the platform developers. First, it would be useful for hardware designers to know the impact of choosing different LP. We suggest a direction by executing a sample stage under different LP configurations. Second, we come up with an application partitioning strategy that makes continuous sensing on mobile devices practical. Third, we propose a proper set of API library as well as runtime design strategy for the hardware platform developer.

Low-Power Processor Selection. We next develop guidelines for selecting an appropriate low-power processor. Fig-



(a) Activity Recognition Application (b) Simplified SoundSense Application (c) Activity Recognition + SoundSense

Figure 5. Multiple Stages Running Together

ure 6 shows the break-even graphs using KissFFT executing on different HW configurations. From the figure, we observe that a) the different HW configurations result in widely varying energy numbers. b) ARM processor has a much larger wakeup delay compared to others, and it adds significant overhead. c) relatively large performance differences between individual LPs, e.g. AVR vs MSP, may have negligible impact on the overall performance when paired with an AP. So, we conclude that any simple low power processor with a small wakeup transition delay is suitable as the LP. On the other hand, feature-rich processors with custom HW support that does not get exercised very often may result in a poor overall performance when paired with an AP.

Paritioning Guidelines. The simulation results in the evaluation section show that most of the energy savings due to partitioning an application between the AP and the LP come from the assignment of more frequently executed sensor sampling and buffering stages to the LP. Beyond that, there is a diminishing return. Even those computing stages that lie well below the break-even curve do not add much to the overall energy savings since they are executed at much slower frequency compared to the sampling stages. Based on these observations, we draw the following conclusion for application partitioning between the AP and the LP.

AP: For tasks found to be more efficient on the AP (above the trade-off curve in Figure 2), always place on the AP since the efficiency of the AP out-weights the transition penalty.

LP: For tasks that are more efficient on the LP, simplest tasks that are executed most frequently within an app should first be placed on the LP since the savings are the greatest for these. From the survey of applications, this almost always includes the sampling and buffering task.

AP and LP: For tasks more efficient on the LP, but executed with lower frequency, (subsequent stages within an application) placement on LP provides only marginal gains in energy (since these tasks are not the dominant energy consumers within the app). However, since these tasks are relatively more complex computationally, they incur an opportunity cost by occupying a disproportionate amount of LP resources that could otherwise be used host the lowest layer and most energy saving tasks of other apps on the LP. Hence, such tasks should be implemented on the LP along with a

Layer	Description	Examples
1	Sampling&Buffering	SampleAndBuffer(Rate, BufSize)
2	Simple processing, Advanced Filtering	GetMean(), GetVar(), GetSIMDAdd()
3	Platform-specific Special libraries	FFT/DFT, MFCC

Table 8. Layered API suggestions for LP

mechanism to evict them from the LP when additional applications are started. One way to achieve this is to have the app developer only develop their app on the AP and allow them to use some of the pre-implemented methods on the LP to service the lowest layer, frequently used, stages in a large class of applications. If a new application is started on the device, and the LP does not have resources to accommodate the lowest layer sampling and buffering stage for the new app, one of the higher layer methods on the LP is evicted and the app uses its own implementation on the AP.

Following the guidelines above, we propose a *layered set of APIs for the LP* based on our survey of a number of applications (Table 8). The list of APIs chosen will suffice for not only the specific apps in Table 3 but also other applications that share similar types of data processing. As new types of apps are envisioned, the list will need to be correspondingly expanded.

The lowest layer API, to sample and buffer data from a sensor, is common to all apps surveyed and found to be universally most efficient on the LP. This method is also often the only app stage that yields significant energy savings by migrating to the LP. Hence we suggest that a sampling and buffering API be provided for each sensor on the LP.

The decision becomes trickier for computations typically used at subsequent stages after sampling. Using the survey of applications and the trade-off analysis in earlier section, along with the measurements performed for multiple apps, we recommend a second layer of APIs to be provided on the LP, listed at layer 2 in Table 8. The set of methods included in layer 2 occupy a relatively small amount of resources on the LP in terms of memory and CPU cycles used at run time and are useful to multiple applications.

Lastly, we propose a third layer of APIs chosen from among the methods measured to be more efficient on the LP and also used among multiple apps. A key distinction from the layer 2 APIs is that these APIs may occupy a significant amount of resource on the LP. Hence, it is critical that for every layer 3 method that is selected to be implemented on the LP, an equivalent version also be provided on the AP. At any time when a new app is started and the LP does not have resources to accommodate its lowest layer task for sampling and buffering, any layer 3 methods running on the LP will be de-activated and their counterparts initialized on the AP, since the energy gain is much higher by making space for the lowest layer methods on the LP.

CONCLUSION

This paper examined how to split multiple modules of a continuous sensing application between the main processor and a low-power processor to reduce the overall energy consumption. To achieve the goal, we first identified the common structure of applications envisioned in the literature. Using this, we modeled and analyzed the system energy consumption. We performed extensive measurements on various combination of LPs and one of the latest APs to identify efficient operating points. We also quantified energy consumption of multiple applications running simultaneously. From the insights derived from these analyses and measurements, we presented important design implications for the platform developers. In summary, assigning simple and frequent sampling & buffering, and arithmetic operations to the LP brings the most energy benefit. Additional tasks can be placed on the LP, but careful dynamic job scheduling based on accurate resource monitoring for LP is needed at runtime for such tasks.

REFERENCES

1. R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. "Simplifying Cyber Foraging for Mobile Devices". In *MobiSys*, 2007.
2. L. Benini and G. d. Micheli. System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5:115–192, April 2000.
3. G. Challen and M. Hempstead. The case for power-agile computing. In *HotOS*, 2011.
4. M.-C. Chiu, S.-P. Chang, Y.-C. Chang, H.-H. Chu, C. C.-H. Chen, F.-H. Hsiao, and J.-C. Ko. Playful bottle: a mobile social persuasion system to motivate healthy water intake. In *Ubicomp*, 2009.
5. T. Choudhury et al. The mobile sensing platform: An embedded system for activity recognition. *IEEE Pervasive Magazine*, 7(2):32–41, April 2008.
6. S. Consolvo, P. Klasnja, D. W. McDonald, D. Avrahami, J. Froehlich, L. LeGrand, R. Libby, K. Mosher, and J. A. Landay. Flowers or a robot army?: encouraging awareness & activity with personal, mobile displays. In *UbiComp '08*, 2008.
7. E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, and P. Chandra, Ranveer an d Bahl. "MAUI: Making Smartphones Last Longer with Code Offload". In *MobiSys*, 2010.
8. V. Devadas and H. Aydin. Real-time dynamic power management through device forbidden regions. In *IEEE RTAS*, 2008.
9. J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay. Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *MobiSys*, 2007.
10. M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *DAC*, 2008.
11. S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *MobiSys*, 2008.
12. J. A. Kientz and G. D. Abowd. Kidcam: Toward an effective technology for the capture of children's moments of interest. In *Pervasive*, 2009.
13. D. H. Kim, Y. Kim, D. Estrin, and M. B. Srivastava. Sensloc: sensing everyday places and paths using less energy. In *SenSys*, 2010.
14. N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell. A survey of mobile phone sensing. *IEEE Communications Magazine*, 48(9):140 –150, sept. 2010.
15. M. L. Lee and A. K. Dey. Lifelogging memory appliance for people with episodic memory impairment. In *UbiComp*, 2008.
16. J. Lester, T. Choudhury, and G. Borriello. A practical approach to recognizing physical activities. In *Pervasive*, 2006.
17. J. Lester, C. Hartung, L. Pina, R. Libby, G. Borriello, and G. Duncan. Validated caloric expenditure estimation using a single body-worn sensor. In *Ubicomp '09*, 2009.
18. F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *ASPLOS*, 2012.
19. H. Lu, A. Bernheim Brush, B. Priyantha, A. Karlson, and J. Liu. SpeakerSense: energy efficient unobtrusive speaker identification. In *Pervasive*, 2011.
20. H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys*, 2009.
21. H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *SenSys*, 2010.
22. T. Maekawa, Y. Yanagisawa, Y. Kishino, K. Ishiguro, K. Kamei, Y. Sakurai, and T. Okadome. Object-based activity recognition with heterogeneous sensors on wrist. 6030:246–264, 2010.
23. E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell. Darwin phones: the evolution of sensing and inference on mobile phones. In *MobiSys*, 2010.
24. P. Mohan, V. N. Padmanabhan, and R. Ramjee. "Nericell: rich monitoring of road and traffic conditions using mobile smartphones". In *SenSys'08*, Nov. 2008.
25. R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *NSDI*, April 2009.
26. N. Oliver and F. Flores-Mangas. Healthgear: a real-time wearable system for monitoring and analyzing physiological signals. In *BSN*, 2006.
27. T. Park, J. Lee, I. Hwang, C. Yoo, L. Nachman, and J. Song. E-gesture: a collaborative architecture for energy-efficient gesture recognition with hand-worn sensor and mobile devices. In *SenSys*, 2011.
28. B. Priyantha, D. Lymberopoulos, and J. Liu. LittleRock: Enabling energy-efficient continuous sensing on mobile phones. *IEEE Pervasive Computing*, 10(2):12–15, Feb. 2011.
29. M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *MobiSys*, 2011.
30. K. K. Rachuri, M. Musolesi, C. Mascolo, P. J. Rentfrow, C. Longworth, and A. Aucinas. Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In *Ubicomp*, 2010.
31. G. Raffa, J. Lee, L. Nachman, and J. Song. Don't slow me down: Bringing energy efficiency to continuous gesture recognition. In *ISWC*, pages 1–8. IEEE, 2010.
32. TI. OMAP 5 mobile application platform. 2011.
33. G. Yavuz, M. Kocak, G. Ergun, H. O. Alemdar, H. Yalcin, O. D. Incel, and C. Ersoy. A Smartphone Based Fall Detector with Online Location Support. In *PhoneSense*, 2010.