View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Apollo

LEO: Scheduling Sensor Inference Algorithms across Heterogeneous Mobile Processors and Network Resources

Petko Georgiev[§], Nicholas D. Lane[†*], Kiran K. Rachuri[‡], Cecilia Mascolo[§]
[§]University of Cambridge, [†]University College London, [*]Bell Labs, [‡]Samsung Research America

Abstract

Mobile apps that use sensors to monitor user behavior often employ resource heavy inference algorithms that make computational offloading a common practice. However, existing schedulers/offloaders typically emphasize one primary offloading aspect without fully exploring complementary goals (e.g., heterogeneous resource management with only partial visibility into underlying algorithms, or concurrent sensor app execution on a single resource) and as a result, may overlook performance benefits pertinent to sensor processing.

We bring together key ideas scattered in existing offloading solutions to build LEO – a scheduler designed to maximize the performance for the unique workload of continuous and intermittent mobile sensor apps without changing their inference accuracy. LEO makes use of domain specific signal processing knowledge to smartly distribute the sensor processing tasks across the broader range of heterogeneous computational resources of high-end phones (CPU, co-processor, GPU and the cloud). To exploit short-lived, but substantial optimization opportunities, and remain responsive to the needs of near real-time apps such as voice-based natural user interfaces, LEO runs as a service on a low-power co-processor unit (LPU) to perform both *frequent* and *joint* schedule optimization for concurrent pipelines. Depending on the workload and network conditions, LEO is between 1.6 and 3 times more energy efficient than conventional cloud offloading with CPU-bound sensor sampling. In addition, even if a general-purpose scheduler is optimized directly to leverage an LPU, we find LEO still uses only a fraction ($< 1/7$) of the energy overhead for scheduling and is up to 19% more energy efficient for medium to heavy workloads.

CCS Concepts

•**Computer systems organization** → **Heterogeneous (hybrid) systems;** *Embedded software;* •**Information systems** → Mobile information processing systems;

Keywords

Mobile Sensing; Offloading; Scheduling; DSP

1. INTRODUCTION

Equipped with powerful processors, multiple radio interfaces, and a variety of sensors, smartphones have revolutionized the mobile application space. A rapidly growing trend among smartphone apps is to enhance functionality and provide advanced features in near real-time using inferences from the phone's sensor data [46]. For instance, Shazam [24] uses the phone's microphone to capture audio and identify the music being played. Further, a plethora of fitness applications [1, 16, 22] use the phone's accelerometer sensor for behavior monitoring, while Apple Siri [6] and Microsoft Cortana [13] use the phone microphone to provide a voice-activated digital assistant.

Fine grained data feeds from sensors are often needed for precise activity inference but the sampling of sensors on phones comes at a high energy cost. Hardware and operating systems have overcome this in various ways; sensor access is often restricted through narrow APIs that sample at predefined rates, cloud offloading is used for inference tasks, avoiding the power hungry CPU, and dedicated hardware is introduced to handle specific sensor algorithms (e.g., the M8 co-processor in iPhones for activity detection through the accelerometer). However, these solutions are often ad-hoc and do not scale: *as sensor-based apps become increasingly popular, it is becoming acutely obvious that new mechanisms are required to effectively manage the resources they consume*.

Often the execution of these apps overlaps as they share a basic set of similar trigger contexts; for example, accelerometer-based apps perform computation under motion (transportation mode detection, activity recognition) and speech processing is triggered by detected voice (speech recognition, speaker/owner identification, conversation analysis). Whereas some functionalities such as keyword spotting are ingrained with the mobile hardware, others such as voice commands become integrated with existing apps to streamline user access to the app's services. Overall, an emerging trend is the concurrent operation of sensor-based apps the execution of which is conditioned on a common set of filters (motion, sound, speech, etc.) applied to the sensor stream.

In this work we re-examine some key ideas scattered in existing computational offloading approaches to answer the question: *can we maximize resource utilization from multiple concurrent sensor apps by a better placement of the underlying algorithms across resources and without compromising application's accuracy and responsiveness*? A variety of capable scheduling/offloading approaches have been proposed [56, 32, 54, 42, 63] but they either have different optimization goals or have not fully addressed the above question in the emerging context of concurrent sensor apps with diverse deadlines running on recent off-the-shelf heterogeneous mobile SoCs. On the one hand, a general purpose offloader such as MAUI [32] does not specifically target the unique demands

of sensor app workloads and may overlook optimization opportunities enabled by the semantics of sensing algorithms not exposed by program structure. SymPhoney [42], on the other hand, is a perfect example of how domain-specific signal processing knowledge can be exploited to efficiently manage resource contention of concurrent sensor apps running on the CPU, but does not take advantage of the heterogeneous mobile hardware. Wishbone [56] provides the skeleton for a sensor net cross-resource data flow partitioning framework but its model is geared towards a different optimization goal (i.e., maximizing throughput) and its solver is too heavy to be invoked frequently in response to dynamically generated sensor events from concurrent apps.

We introduce LEO – a purpose-built sensing algorithm scheduler that targets specifically the workloads produced by sensor apps. LEO builds upon the solid body of related work to demonstrate that further increases in the efficiency of sensor processing *without reducing the app accuracy* can be achieved by bringing together 4 key ideas scattered across existing systems: 1) full leverage of heterogeneous hardware, 2) joint sensor app optimization, 3) frequent schedule re-evaluation, and 4) exposing algorithm semantics to the optimization model. As a result, LEO is able to optimally offload individual stages of sensor processing across the complete range of heterogeneous computational units (viz. the co-processor, CPU, cloud, and provisionally a GPU). A key innovation in LEO is that all offloading decisions can be performed on the smartphone co-processor, this enables scheduling to be low energy and frequently performed. Consequently, LEO adjusts how sensor algorithms are partitioned across each computational unit depending on fluctuating resources (e.g., network conditions, CPU load) in addition to which sensor apps are executing concurrently. With LEO, the energy and responsiveness trade-offs of sensing algorithms of all apps are *jointly* optimized, leading to more efficient use of scarce mobile resources. The contributions of our work include:

- A scheduler designed to run frequently at a low cost on a low power unit (LPU), with $< 0.5\%$ of the battery daily. The scheduler optimizes sensor inference algorithm offloading across multiple concurrently running apps with diverse timeliness requirements without sacrificing inference accuracy.
- A comprehensive proof-of-concept implementation built on a smartphone development board [20]. Our prototype includes an extensive library of sensing algorithms for feature extraction and classification (including DNNs [40], GMMs [27] etc.) needed for common forms of context inference. The library is rich enough to support numerous apps from the literature – as a demonstration, we implement 7 recently proposed systems (e.g., stress detection, activity recognition, spoken keyword spotting).
- A systematic evaluation of LEO's overhead and schedule optimality as well as a thorough analysis of the system energy savings under a variety of network conditions and workloads. Compared to a principled general-purpose offloader that leverages an LPU in addition to cloud, LEO requires about 7 to 10 times less energy to build a schedule, and still reduces the energy consumption by up to 19% for medium to heavy sensor workloads.

2. SENSOR APPS ON MOBILE DEVICES

LEO exclusively targets sensor apps that are characterized by their need to sample and interpret sensors present in smartphones. Here, we describe key varieties of sensor apps and overview typical sensing operations. Table 1 details examples that are either research prototypes or commercially available.

Anatomy of a Sensor App. Every sensor app includes specialized code responsible for sensor data sampling and processing, distinctly different from the app specific logic. LEO is designed to efficiently execute this sequence of sensor processing stages. Irrespective of purpose, the dataflow of sensor processing within these apps share many similarities. Processing begins with the sampling of sensors (e.g., microphone, accelerometer). Feature extraction algorithms are then used to summarize collected data (as a vector of values), the aim is for these features to describe the differences between targeted behavior (e.g., sleep, or running) or context. Identifying which activity or context is present (i.e., to make an inference) in the sensor data requires the use of a classification model. Models are usually built offline prior to writing a sensor app based on examples of different activities. Although inference is the most fundamental sensor operation performed in sensor apps, considerable post-inference analysis is often needed (e.g., mining sleep or commute patterns.)

Applications	Sensor	Purpose
RunKeeper [22] Accupedo Pedometer [1]	Accel	activity tracking
Shake Gesture Library [23]	Accel	gesture commands
Hot Keyword Spotting [29]	Mic	voice activated services
Shazam [24]	Mic	song recognition
SocioPhone [47] SpeakerSense [50] Crowd++ [66], SocialWeaver [53]	Mic	conversation context
EmotionSense [61] StressSense [51], MoodScope [48]	Mic	emotion recognition
Siri [6], Cortana [13]	Mic	digital assistants
Waze [26], Moovit [15]	GPS	traffic monitoring

Table 1: Example Sensing Apps.

Sensor App Workloads. Just like conventional apps, different combinations of sensor apps are continuously executed. There are two dominant usage models, *continuous sensing* and *triggered sensing*, each with differing user expectations of responsiveness and exerting differing types of energy strain.

Continuous Sensing. Most apps of this type are "life-logging" [36] and are commonly used to quantify daily user routines. They aim to capture data from the phone throughout the day: this demands their sensing sampling and processing algorithms to be extremely energy efficient. In contrast, because of the focus on long-run behavior they can often tolerate large processing delays; for example, users may review data at the end of the day and the analysis is indifferent to events from the last few minutes.

Triggered Sensing. This category includes sensor apps that are initiated either explicitly by the user or by an event. Examples are apps started by the user to monitor a meeting (Social Weaver [53]) or a workout (Run Keeper [22]). Users often need sensing to be completed during the event in near real-time (e.g., to gouge their effort during an exercise routine to determine if more effort is required). Sensing apps can also be started in response to an event. For instance, a smartphone may need to determine the current ambient context in response to an incoming call to decide if it should be sent straight to voice mail as the user might be driving. These types of sensing apps have a much higher need to be responsive than continuous sensing ones; but as they are often comparatively short lived, energy restrictions may be relaxed to achieve processing deadlines. A known approach to reduce the high energy coming from the continuous trigger logic evaluation is to instrument apps to use a sensor hub where irrelevant readings are filtered automatically [63].

A key unanswered challenge today is how to maximize the resource efficiency for this diverse and dynamic sensing app workload, while maintaining other phone services, e.g., email, music and games, functioning.

Application	CPU Latency	DSP Latency	CPU Energy	DSP Energy
Activity Recognition [52]	0.002s	0.006s	3mJ	0.4mJ
Speaker Counting [66]	0.533s	1.279s	794mJ	86mJ
Emotion Rec. (14 GMMs) [61]	2.410s	8.941s	4073mJ	340 mJ
Speaker Id. (23 GMMs) [61]	3.673s	13.325s	6208mJ	506 mJ
Stress Detection [51]	1.623s	3.115s	2580mJ	174mJ
Keyword Spotting [29]	0.720s	2.249s	1152mJ	113mJ

Table 2: Application pipeline latency and energy compared on the CPU and DSP (default clock frequency). Results are obtained with a Monsoon Power Monitor [14] attached to a Snapdragon 800 Mobile Development Platform for Smartphones [20].

3. LIMITS OF SENSOR APP SUPPORT

Developers of sensor apps today work with black-box APIs that either return raw sensor data or the results of a limited selection of sensing algorithms [4, 12]. The underlying resources (e.g., emerging low power co-processors or system services that regulate sensor sampling rates) that feed these APIs are closed and inaccessible to developers. Critically, because the mobile OS lacks the necessary mechanisms to regulate the energy and responsiveness trade-offs of sensing algorithms, how these apps share resources, remains unoptimized. We now describe the limits of co-processors and cloud offloading support of sensor apps.

Low Power Co-Processor – An Underutilized Resource. Prior to the advent of co-processors, the CPU was used for both sensor sampling and data computation. This resulted in unacceptable energy trade-offs that made many sensing scenarios impractical.

Potential Energy Savings. To illustrate potential implications for sensing, we perform an experiment with a special development-open version of the Qualcomm Snapdragon 800 System on Chip (SoC) [21], shipping in phones (e.g., Nokia Lumia and Samsung Galaxy) [25]). Table 2 compares the energy and latency of a range of sensor processing algorithms (Section §6) for the accelerometer and microphone on the DSP and the CPU of the Qualcomm SoC. We observe an overall reduction of 8 to 15 times in energy consumption to a level at which it becomes feasible for smartphones to perform various sensing tasks continuously.

However today's smartphone co-processors cannot fully address the needs of sensor app workloads because of two critical limitations: (1) APIs remain narrow and inflexible; and (2) the DSPs are closed.

Limited APIs. Sensor engine APIs similar to the ones provided by Apple [5] and Nokia [12] enable a variety of location and physical activity related sensor apps. Yet, the algorithms needed for many other sensors uses such as custom gesture recognition or fall detection are absent. Unless the developer's use case is already supported by the APIs, CPU-based sampling and processing must be used. While APIs supporting more apps are likely to be offered in future, the closed APIs also prevent stages of sensor processing to be divided between the co-processor and other units like the CPU. Without this ability only algorithms simple enough to be run solely on the co-processor can be supported.

Closed to Developers. There are two main reasons behind why co-processors are closed. First, embedded components such as co-processors are easily overwhelmed if not carefully utilized. Opening the co-processor requires complex new OS support providing, for example, a multi-programming environment (i.e., concurrent sensor apps) in addition to isolating apps so that excessive resource consumption by one app would not compromise others. Second, an open co-processor requires developers to engage in embedded programming. This significantly increases development complexity, requiring code for each computational unit (DSPs, CPU) and forcing greater hardware awareness.

Is Cloud Offloading Alone the Solution? Although cloud offloading can enable significant reductions in latency [69, 30, 32, 60], just like existing use of co-processors it is unable to fully meet the needs of sensor app workloads, for two primary reasons – sensitivity to network conditions and CPU-bound offloading.

Network Conditions. Under good network conditions (e.g., low RTTs, typical 3G/WiFi speeds) offloading sensor processing, like face recognition, can result in energy and latency improvements of 2.5 times [32]. But such conditions are not always present. For example, a survey of more than $12,000$ devices worldwide [65] finds that a sizable 20% of the devices are not exposed to 3G, LTE or WiFi connectivity at least 45% of the time.

CPU-bound Offloading. Conventional offloading applied to mobile sensing (e.g., [57]) must rely on the CPU for local operations. CPU-based sensing algorithms are highly energy inefficient. As a result, cloud offloading is severely constrained in the variety of sensor apps to which it can be applied. For example, apps that require continuous sensing cannot be supported with offloading alone. Even the emergence of co-processor support in smartphones has not addressed this problem. Because current co-processors only provide the end result of sensor processing (e.g., an inference), they are unable to act as the front-end to a chain of operations that includes stages executed remotely.

As we have seen, neither co-processors nor cloud offloading fully address the needs of a sensor app workload. What is needed are not additional ad-hoc optimization approaches. Instead, a system service is required that has visibility of the sensor algorithms being executed in each app, along with access to the full range of computational units and other resource types available to the device.

4. LEO OVERVIEW

Towards addressing the shortcomings of general computational offloading for sensor app workloads, LEO is a sensor algorithm scheduling engine that maximizes the energy efficient execution of a variety of sensor apps. LEO is used by developers via a set of Java/C APIs with a unified interface (see §6 for details). Through the Java API developers can specify the sequence of sensing algorithms (e.g., feature extraction, classifier model) required by their app to collect and process data. In turn, LEO leverages the internal structure of the sensing algorithms predefined in our library of algorithmic components to partition the execution of each algorithm across all available computational units (even conventionally closed DSPs found in smartphone SOCs, along with the CPU, cloud, and a GPU). Because of the rich portable library of sensing algorithms (ranging from DNNs to domain-specific features for emotion recognition) LEO is able to support a wide range of sensor processing (and thus apps).

LEO re-examines several concepts related to scheduling/offloading from previous systems and re-evaluates them in the context of concurrent sensor apps running on off-the-shelf heterogeneous mobile SoCs. LEO shows that we do not need to compromise the utility/accuracy of applications or sacrifice their responsiveness in order to gain substantial energy savings. Instead, *maximizing resource utilization can be performed by a smarter distribution of concurrent sensor algorithms with well known semantics across multiple resources*. To achieve this, LEO:

1) considers offloading decisions collectively for heterogeneous mobile processors and cloud. LEO builds upon related work such as Wishbone [56] and MAUI [32] to solve a Mixed Integer Linear Programming (MILP) global resource optimization problem that directly targets energy efficiency as its objective.

2) jointly optimizes resource use for multiple apps simultaneously. This promotes cooperation in using network bandwidth or

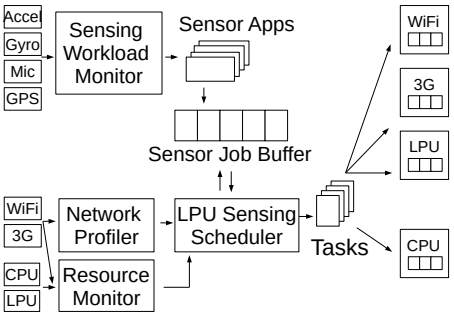


Figure 1: LEO architectural components.

scarce co-processor memory/computation. As a result, maximizing resource use is done across the full sensing ecosystem rather than leaving individual apps do guesswork on when resources are busy.

3) exposes the internal structure of the pipeline stages to the offloading engine for fine-grained energy control. LEO provides a rich set of reusable algorithmic components (feature extraction, classification) which are the building blocks of common sensor pipelines. By leveraging sensor processing specific knowledge LEO decomposes pipelines into more granular units, orchestrates and interleaves their execution even under tight latency constraints by working on multiple smaller tasks in parallel.

4) frequently re-evaluates the schedule to remain responsive to sensor processing requests. Sensor apps generate mixed workloads with near real-time and delayed execution requirements. To provide timeliness guarantees while coping with changes in network conditions and bursts of generated sensor events such as detected speech, noise, or motion, LEO computes fast, reactive schedules that are frequently revised. A key enabler for this is the ability of LEO to run as a service on one of the hardware threads of the low power DSP where the scheduler solves heuristically the global optimization problem mentioned above.

4.1 Architectural Overview

In Figure 1 we show a birds-eye view of the system architecture and its operational worklfow. The system supports a mixture of near real-time and delay-tolerant sensor processing apps. The pipeline stages of these apps are typically triggered by sensing events such as the detected presence of speech from the sensor data streams, or by logic embedded in the sensor app. Example applications with their trigger contexts are recognizing emotions from voice, or counting the number of steps when the user is walking. Over time and as sensor events are encountered, the apps generate *job* definitions which are buffered requests for obtaining an inference (e.g., detected emotion) from the sensor data. Periodically, the sensor offloading scheduler, known as *LPU Sensing Scheduler*, inspects the sensor job buffer for the generated workload of sensor processing requests, and makes scheduling decisions that answer the questions: 1) How should the pipelines be partitioned into sensor tasks? and 2) How should these tasks be offloaded if needed?

Sensing Workload Monitor. A set of binary filters (e.g., "silence vs. noise", "speech vs. ambient sounds", "stationary vs. moving") comprise the *Sensing Workload Monitor* which continuously inspects the sampled sensor data on the Low Power Unit (DSP in our case) for the presence of relevant events (for triggered sensing apps). Once such events are detected or in response to the sensor app, job requests are placed in a global queue that buffers them together with the raw sensor data.

LPU Sensing Scheduler. This component represents the core scheduling algorithm that decides how the pipeline execution should be partitioned into sensor tasks and where these tasks should be executed (LPU, CPU, cloud or potentially GPU). The scheduler inspects the sensor job buffer once every t seconds for processing requests where t is a configurable system parameter currently set to a short window of 1 second. Queued tasks are periodically scheduled and executed before the next period expires. Critically, LEO defines a mathematical optimization problem that can be solved frequently and maintains a short rescheduling interval in order to systematically re-evaluate fleeting optimization opportunities and remain responsive to apps such as voice activation services with near real-time requirements. The high levels of responsiveness and frequent on-demand optimizations are largely enabled by two key design choices. First, the scheduler reorganizes the structure of sensor pipelines to create more modular processing tasks via three key techniques detailed in §5.1: *Pipeline Partitioning*, *Pipeline Modularization*, and *Feature Sharing*. Second, the scheduler employs a fast heuristic optimization solver (based on metaheuristics) that is executed with an ultra low overhead on the LPU to find a near optimal assignment of tasks to resources.

Resource Monitor. A *Resource Monitor* provides feedback to the LPU Sensing Scheduler with regard to changing CPU load or network conditions such as connecting to or disconnecting from a WiFi network.

Network Profiler. Similarly to MAUI [32], a *Network Profiler* sends a fixed 10KB of data and measures the end-to-end time needed for the upload. Fresh estimates are obtained every time the scheduling engine ships data for processing to a remote server. To keep measurements fresh, profiler data is sent every 15 mins in case no offloading has been done.

Offline Profiling. Last, an offline app profiler obtains estimates of the energy consumption and latency for each of the application pipeline stages (feature extraction and classification) measured on the CPU, LPU, and for some algorithms on the GPU. The measurements serve as an input to the LPU Sensing Scheduler that distributes sensor pipeline tasks across offloading resources. The profiling session is a one-time operation performed offline since the mobile OSs have limited APIs for performing fine grained energy measurements [32] and only report the percentage of the battery left, largely insufficient to cover the profiling needs.

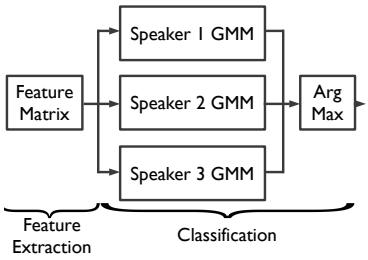
5. LEO DESIGN COMPONENTS

LEO is designed to manage the offloading needs of sensor apps with both near real-time and delayed deadline requirements. In this section we detail how LEO leverages algorithm semantics to optimize resource use, and also formally define the optimization problem that jointly decides for concurrent apps how to execute their algorithms across resources.

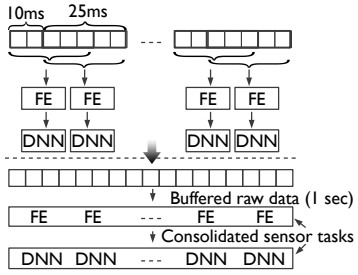
5.1 Restructuring Sensor App Execution

Pipeline Partitioning. Sensing pipelines are decomposed into logical chunks of computations to increase the granularity of the sensor tasks and enable their more comprehensive exposure to the offloading components (cloud, LPU, and GPU). This can potentially lead to more efficient local-remote splits and parallelize execution across multiple resources to meet the tighter deadlines of near real-time applications.

Pipelines are divided into two types of sensor tasks: 1) *feature extraction*, typically represented by a single task per application; and 2) *classification*, which may be further decomposed into multiple identical tasks the output of which is combined by a simple aggregation operation (e.g., *max*, *sum*). For instance, the inference stage of a typical Speaker Identification application [61] is usually organized around multiple class models (Gaussian Mixture Models



(a) Pipeline Partitioning



(b) Pipeline Modularization

Figure 2: Sensor pipeline restructuring techniques.

[27]) as shown in Figure 2a, one for each speaker. During the classification process, the extracted audio features that summarize the acoustic observations are matched against each model in turn. The speaker, the model of which with highest probability corresponds to the features, is the end output of the pipeline.

The ways in which signal processing algorithms can be partitioned is predefined in our library. At runtime LEO decides whether and how to leverage the partition points by solving a global resource allocation problem and managing cross-resource communication with custom implemented message passing. As a result, applications that use the algorithms we offer automatically benefit from efficient task distribution without them knowing the details of the pipeline execution plan. Further, the decomposition into shorter duration tasks enables the pipeline stages of concurrent apps to be interleaved – the result being higher utilization of the energy efficient but memory-constrained LPU.

Pipeline Modularization. Partitioning the pipelines into their natural processing stages increases the granularity at which sensor apps operate. However, depending on the application subject to partitioning, the technique may sometimes produce a large number of sensor tasks that unnecessarily pollute the resource optimization space with a forbiddingly high number of offloading configurations. *The goal of the Pipeline Modularization is to consolidate multiple sensor tasks generated at a high frequency into a single modular computational unit processed at a reduced frequency.*

A Keyword Spotting application based on Deep Neural Nets (DNNs) [29], for example, as shown in Figure 2b extracts features from 25ms frames, and performs classification (neural network propagation and smoothing) on a sliding window of 40 frames every 10ms. In other words, every second the sensor app generates 100 classification and feature extraction tasks. This high frequency of computations enables the app to maintain almost instantaneous responsiveness to detected hot phrases. However, at a small latency price we can reduce the amount of tasks a hundredfold if we group all feature extractions or classifications in a second into a modular unit by buffering the sensor data and performing them together on the accumulated speech samples. Thus, processing is performed at a reduced frequency once every second, greatly simplifying the search for the optimal task allocation, while at the same time still maintaining near real-time responsiveness of the app to up to 2 seconds after the detection of a hot phrase.

Feature Sharing. The pipeline decomposition allows us to register modular identifiable tasks into the queue of sensor processing requests. Each feature extraction task is identified by a reference to a position in the sensor stream and the type of features to be extracted from the raw data. This allows LEO to detect overlapping computations and eliminate redundancies when multiple applications require the same set of features for their stages deeper into the pipeline. One example of shared features are the Perceptual Linear Predictive Coefficients (PLP) needed by both Speaker Identification and Emotion Recognition applications [61].

5.2 LEO Optimization Solver

The LEO solver uses the restructured pipeline components as well as data collected by the Sensor Workload Monitor and Network Profiler as an input to a global joint optimization problem (similarly to MAUI[32] and Wishbone [56]) that determines which and where sensor app tasks should be offloaded. Unlike Wishbone, the original formulation of which targets throughput and minimizing node-to-server communication overhead, LEO solver's goal is to find a multi-app partitioning strategy that minimizes the mobile device energy consumption subject to latency constraints.

The solver takes advantage of the pipeline reorganization techniques introduced in the previous subsection to generate modular sensor task definitions with *loose data dependencies*: feature extraction output serves as input to classification and tasks from the same pipeline stage can be computed in parallel across units. Formally, the LPU Sensing Scheduler solves a mixed integer linear programming problem (MILP) with relaxed data dependencies constraints. The objective is to minimize the total energy consumption for processing all sensor tasks generated in a time window τ by all actively running sensor apps:

$$\text{Min} \quad \sum_{i,q,u} x_{iqu} e_{iqu} + \sum_{i} F_{mem}(i, x_{iq\xi}) w_{uplink} p_{\xi} \qquad (1)$$

where

- x_{iqu} denotes the total number of computations from application's i pipeline stage q that will be processed on computational unit $u \in$ {CPU, LPU, GPU } (or networked resource ξ when $u = \xi \in$ {3G, WiFi, Bluetooth}).
- e_{iqu} indicates the energy consumed for performing these computations on the resource u ($e_{iq\xi} = 0$).
- $F_{mem}()$ is a linear function mapping the number of remotely executed sensor tasks to the size of the application data needed for network transfer.
- w_{uplink} – most recently estimated uplink speed (Kbps).
- p_{ξ} is the estimated average power in mW for performing the network transfer.

The objective expresses the estimated total energy expenditure from performing computations on the assigned offloading resources plus the energy needed to transfer data for remote execution. The offloading schedule is subject to the following constraints:

- The total execution time for processing tasks locally (Equation 2) and remotely (Equation 3) must not exceed the time window τ:

$$\text{s.t.} \ \ \forall u \ \sum_{i,q} x_{iqu} t_{iqu} \leq \tau k_u \qquad (2)$$

$$\text{s.t.} \sum_i F_{mem}(i, x_{iq\xi}) w_{uplink} + \sum_{i,q} x_{iq\xi} t_{iq\xi} \leq \tau \quad (3)$$

Here t_{iqu} is the time in seconds required by computation of type q to be performed on computational unit u, and k_u is the number of concurrent threads on resource u.

- The total number of sensor tasks offloaded across resources must be equal to the number of tasks $n_{iq}(\tau)$ generated by the buffered processing requests in time window τ.

$$\text{s.t.} \sum_u x_{iqu} = n_{iq}(\tau) \quad (4)$$

We note that the typical restructured pipeline computations from our representative examples can be easily executed with subsecond latencies which enables us to shrink the offloading window τ to 1 second. This also helps with fast reactive dispatching of computations that require tight timeliness guarantees (e.g., Keyword Spotting). In our implementation, although the tasks from apps that do not need near real-time requirements are scheduled in a batch with other tasks under tight 1-second constraints, their actual execution is postponed and rescheduled at a later stage if executing them before the next rescheduling interval expires means that the power-hungry CPU will be used.

5.3 Running the Solver on the LPU

The optimization problem defined in the previous section would typically be solved with standard optimization tools such as GLPK [7] or *lp_solve* [11]. However, we observe that the underlying algorithm that systematically explores the scheduling configurations to find the optimal solution is too heavy to be performed frequently. In fact, when we set the time window for buffering processing requests to 30 seconds, and increase the number of scheduled applications to 9, the algorithm takes seconds to minutes to complete even on the quad-core Snapdragon CPU. For our aims the general-purpose solver scales poorly with the increase in number of applications and processing requests. Instead, we adopt a heuristic algorithm that can be run efficiently on the LPU to constantly re-evaluate a near optimal offloading schedule. We sacrifice the absolute optimality for substantial reductions in the scheduling overhead both in terms of energy and latency.

Heuristic Algorithm. The concrete framework we use is based on *memetic algorithms* (MAs) [55, 31] which are population-based metaheuristics that are highly scalable, are able to incorporate domain-specific knowledge, and are less prone to local optima. However, the scheduling algorithm is not restricted to a concrete choice as long as it conforms to several criteria: it is fast (preferably with polynomial complexity to the number of sensor apps and resources), it finds solutions that are close to optimal, and it is deployable on the LPU. We experimentally find that the memetic algorithm is one that satisfies all these requirements.

The algorithm takes as input sensor pipeline tasks, available offloading resources, and the constraints listed in §5.2 that define the feasible solutions. Algorithm 1 outlines the pseudo code of our heuristic. The basic structure of the algorithm is an iterative procedure that maintains a *population* of candidate schedules the quality of which improves over time and is measured through a *utility* function (our objective defined in Equation 1). Each offloading configuration is represented as shown in Figure 3 and each cell in the table corresponds to the value of the decision variable x_{iqu} defined in our problem statement 5.2.

The memetic algorithm defines a series of schedule transformation (mutation and local search) and recombination (crossover) operations that are applied systematically to the population of schedules updated each iteration. The iteration consists of creating candidate offspring schedules (line 5) from selected parents and subsequently choosing the surviving set of schedules (line 10) that will constitute the next generation. Parent schedules are selected for reproduction with a probability proportional to their utility. Reproduction is performed through a standard two-point crossover operation illustrated in Figure 4. Once offspring schedules are generated, two key transformations are applied to each child (lines $6-9$): mutation to promote diversity, and local search to increase utility. The local search step improves the utility of a newly produced offspring schedule by searching for better neighbor solutions and replacing the original schedule with the neighbor the fitness of which is highest. Finally, the best s schedules survive through the next iteration. To reduce the runtime of the algorithm we limit the number of generations n to 100 and perform the local improvement step every other generation. We resort to standard textbook parameters of the algorithm [31]: population size of 10, 5 parents, 20 child schedules, and a mutation probability set to 0.1.

Algorithm 1 LPU Sensing Offloader Approx. Algorithm

Require: Number of generations n, mutation probability α
1: **function** HEURISTICSEARCH(n, α)
2: $P \leftarrow$ InitialSchedulePopulation()
3: $x \leftarrow$ SelectBestSchedule(P)
4: **for** $i \leftarrow 1$ to n **do**
5: $O \leftarrow$ GenerateOffspringSchedules(P)
6: **for** $c \in O$ **do**
7: $c \leftarrow$ Mutate(c, α)
8: **if** $i\%2 = 0$ **then**
9: $c \leftarrow$ LocalImprovement(c)
10: $P \leftarrow$ SelectNextGenerationSchedules($P \cup O$)
11: $x \leftarrow$ SelectBestSchedule($\{x\} \cup P$)
12: **return** x

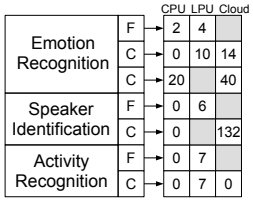


Figure 3: Sched. representation

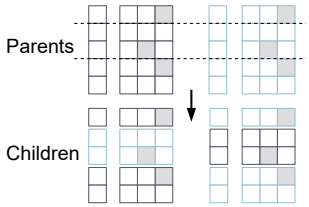


Figure 4: Two-point crossover

6. SYSTEM IMPLEMENTATION

Here we discuss the implementation details of the system prototype and example sensing algorithms used in the evaluation. The development is performed on a Snapdragon 800 Mobile Development Platform for Smartphones (MDP/S) [20] with an Android Jelly Bean OS. As an example LPU we use the Qualcomm Hexagon DSP which is accessible through the C-based Qualcomm Hexagon SDK [19] (of which we use version 1.0.0). Provisional GPU support is added for a subset of the sensor processing algorithms where we use the Adreno 330 GPU which we program with OpenCL 1.1e [17] via the Adreno SDK [2].

Application Model. Similarly to ORBIT [54] we adopt a component-based programming model where we provide an extensive library of reusable signal processing algorithms which developers can use to build and integrate their sensing pipelines into application code. We implement 7 domain-specific categories of feature extraction algorithms and 5 popular machine learning models covering the narrow waist of computations for multiple apps from the mobile sensing literature. With this library we have prototyped the

Application	Sensor	Description	Main Features	Inference Model	Frame	Window
Activity Rec. [52]	Accel	walking, running, etc.	Freq. and Time Domain	J48 Decision Tree	4s	4s
Step Counting [28]	Accel	step counting	Time Domain	Windowed Peak Thresholding	4s	4s
Speaker Count [66]	Mic	speaker counting	MFCC [35], pitch [33]	Unsupervised Clustering	32ms	3s
Emotion Rec. [61]	Mic	emotion recognition	PLP [39]	14 Gaussian Mixture Models [27]	30ms	5s
Speaker Id. [61]	Mic	speaker identification	PLP	22 Gaussian Mixture Models	30ms	5s
Stress Detection [51]	Mic	stress from voice	MFCC, TEO-CB-AutoEnv [70]	2 Gaussian Mixture Models	32ms	1.28s
Keyword Spotting [29]	Mic	hotphrase recognition	Filterbank Energies	Deep Neural Network [40]	25ms	1s

Table 3: Implemented example sensing applications. The window shows the amount of time features are accumulated from frames before an the classification/inference stage is triggered. Frame lengths shown are the default used in the original works.The used sensor sampling rates are 50Hz for the accelerometer and 8kHz for the microphone.

```
/* Java pipeline specification:
sequence of transforms
applied to the sensor stream */
Pipeline p = new AudioPipeline();
p.trigger(new FrameTrigger("Speech_trigger"));
p.apply(new WindowTransform("PLP_windowFeatures")
    .frameSize(240).frameRate(80).window(500))
 .apply(new ParallelTransform(new Transform[] {
        new GMMTransform("speaker1.gmm"),
        new GMMTransform("speaker2.gmm")
        }, AggregatorType.Argmax));

/* DSP compatible C signature conventions */
// initializes a struct from the memory
// referenced by ptr
void* X_init(int8_t* ptr, void* params);
// amount of memory in bytes (size of state struct)
uint32_t X_getSize(void* params);
// trigger function (e.g., speech detection)
int X_trigger(int16_t* buffer, int size);
// feature extraction
void X_windowFeatures(void* me, int16_t* buffer,
        int size, float** outData,
        int nRows, int nCols);
// inference/classification
float X_windowInference(void* me, float** inData,
    int nRows, int nCols);
```

Figure 5: Example code snippets showcasing the APIs supported by LEO. The Java API can be used to define the structure of a pipeline. The C API conforms to the set of conventions given by the Qualcomm Elite SDK for audio processing on the DSP. The Java pipeline definition is mapped to a set of DSP compatible C routines.

2 accelerometer and 5 microphone sensing applications listed in Table 3. One of the major advantages of resorting to a library-based approach is that we have full control over how the various signal processing tasks can be decomposed, i.e. we can expose the sensing algorithms to our pipeline restructuring techniques (§5.1) without involving the developer in the process. Instead, LEO fully automates the partitioning decisions at runtime.

The algorithms are subject to our pipeline reorganization techniques so that all application pipelines are partitioned as discussed in §5.1, and the Keyword Spotting app is restructured by the Pipeline Modularization technique. Feature Sharing is enabled for the Emotion Recognition and Speaker Identification applications. Further details on the algorithm implementations can be found in the original publications cited in Table 3.

APIs and Accessibility. To enable app components to run on heterogeneous computational resources (CPU, DSP, cloud), all accelerometer and audio processing algorithms are implemented in C with a unified interface following the guidelines of the Hexagon SDK. In Figure 5 we provide a subset of the C signature conventions the various audio processing methods must comply to in order to take advantage of LEO's partitioning capabilities. We maintain the same copies of the C code on the DSP, CPU and on the server. To facilitate the integration of the signal processing components with Java application code we have built a Java Native Interface (JNI) bridge library that interfaces between the CPU and DSP. Further, we have defined a high-level Java API with some notions borrowed from Google Cloud Dataflow programming model [8] (applying a sequence of transforms to the input stream) that can help developers specify custom pipelines built from the reusable components we provide. Sample code defining a speaker identification pipeline with 2 speaker models and a voice-activation trigger is given in Figure 5. The high level components ultimately map to the C functions with conventionalized signatures – e.g., the WindowTransform class accepts as an argument to its constructor the name of the C method ($PLP\_windowFeatures$) that will be invoked to process accumulated raw audio data. Developers can additionally define their custom algorithm components by providing a C implementation that conforms to the conventions we follow.

The wrapper functionality defines the structure of a sensor pipeline which consists of two primary computation types: 1) context triggers, added by the *trigger* method and executed as an optional first processing step; and 2) a series of transforms (feature extraction, classification) chained through the *apply* method so that the output from one transform serves as the input to another. Transforms are of two primary subtypes: primitive and composite. Primitive transforms are executed on a single computational unit as an atomic non-preempted operation. Composite transforms consist of multiple primitive transforms: currently we support a parallel transform operation that allows multiple transforms to work concurrently with the same input on different computational units. The pipeline specification is a one-time operation needed to register the type and sequence of C methods that will be scheduled for execution on the various computational units.

Offloading Service. The DSP has three hardware threads acrhitected to look like a multi-core system. LEO runs continuously on the DSP as a service where we reserve one thread for the offloading decision logic. The pipeline stages of the sensing algorithms (additionally extracted features and classification) are executed in the other two threads. The CPU-DSP interactions (loading model parameters, sharing data) are facilitated through the FastRPC mechanism supported by the Hexagon SDK. Data is transferred via contiguous ION/RPCMem memory buffers carved out into Android OS memory and mapped to DSP memory. Threading is orchestrated via POSIX-style APIs provided by the DSP RTOS (real-time OS) knows as QuRT [18]. An Android background service task that is running on the CPU waits for the DSP to return a list of sensor tasks with assigned resources. If the CPU is in sleep mode, it is woken up by the DSP through a return from a FastRPC call to manage the assignment of tasks to other resources (cloud or GPU).

The schedule evaluation is timed periodically every t seconds (currently $t = 1$), with the primary unit of time being the length of audio frames. LEO accumulates raw sensor data in circular buffers,

filters the type of data based on the registered triggers (motion, speech) and runs the scheduling algorithm every n-th frame (e.g., every 34-the frame when its length is 30ms). The raw data buffers accumulate sensor samples over the largest registered window of processing for the corresponding sensor (e.g, 5 seconds for the microphone). The registered context triggers are typically short-duration routines (less than 5ms per 30ms audio frame, and less than 1ms per 4s accelerometer frame) and we interleave their execution with the schedule evaluation in the same thread.

Offline Training. We expect developers to train their machine learning models offline and provide the model parameters required by our library of classification algorithms in the form of text files which we parse when an application is started/accessed for the first time. For instance, the Gaussian Mixture Models are encoded in the format defined by the widely used in speech analysis HTK toolkit [10]. Our parsing utility is written in C++ so that it can supply the model parameters directly in the JNI bridge.

Provisional GPU Support. We have implemented parallel versions of the two heaviest sensor processing algorithms (the GMM inference of the Speaker/Emotion Recognition and DNN propagation of the Keyword Spotting) in OpenCL [17] with the help of the Qualcomm Adreno SDK [2]. Enabling GPU support requires incorporating additional energy into the scheduling objective function that captures the overhead of approaching the GPU (setting up computation and transferring buffers from the CPU host to GPU device memory). Further, custom algorithms not included in LEO's library would require additional programmer effort to provide an OpenCL implementation in addition to the DSP-compliant C version used by default. Interfacing with the GPU is mediated through the background CPU service which flattens the matrix of features required by the heavy classification stages into contiguous blocks of memory that can be copied with OpenCL commands to GPU device memory.

7. EVALUATION

In this section we evaluate LEO's overhead, the energy improvement over baseline offloading and the energy consumption when running example sensing applications under common smartphone usage and varying network conditions. By default, we use the base version of LEO that handles the heterogeneous processing units with full algorithm support and unified C programming model (CPU, LPU and cloud). We discuss the implications of incorporating the GPU as an extra resource in a separate subsection. The main findings are:

- The cross-app scheduling decisions made by LEO are on average 5% worse than those made by an "oracle" knowing the optimal offloading of tasks to resources.
- LEO is between 1.6 and 3 times more energy efficient than off-the-shelf CPU-based APIs enhanced with conventional cloud offloading of classification tasks. Compared to a general-purpose MAUI-style [32] offloader enhanced to use the DSP, LEO requires only a fraction of the energy ($< \frac{1}{7}$) to build a schedule and is still up to 19% more energy efficient for medium and heavy workloads.
- The overhead of the LPU Sensing Scheduler is low ($< 0.5\%$ of the battery daily), allowing frequent rescheduling.
- Considering the smartphone daily usage patterns of 1320 Android users, in more than 90% of the cases LEO is able to operate for a full day with a single battery charge with other applications running on the phone, while providing advanced services such as keyword spotting, activity and emotion recognition.

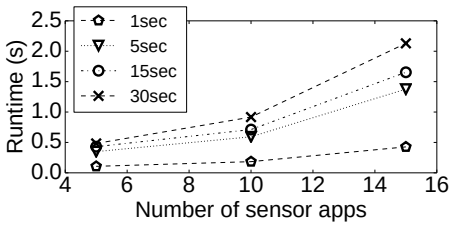


Figure 6: Runtime of the scheduling algorithm as a function of the number of apps and the length of the rescheduling interval.

7.1 Baselines Definition

Here we introduce commonly found scheduling/offloading alternatives that we are used to compare the performance of LEO.

- *DSP+Cloud.* This strategy uses the DSP for feature extraction and then ships the features to a remote server for processing.
- *CPU+Cloud.* This is an alternative that follows a conventional cloud offloading strategy where features are extracted locally on the CPU and then sent to the cloud for classification.
- *Greedy Local Heuristic.* A greedy strategy offloads sensing tasks locally on the mobile device by first pushing to the DSP the CPU expensive computations. The tasks are sorted in descending order of their CPU-to-DSP energy ratio so that those with the largest *energy gain* factors are prioritized for execution on the DSP.
- *Delay-Tolerant.* To demonstrate the huge performance boosts from delayed sensor inference execution, we provide a delay-tolerant version of LEO that runs the optimization solver with relaxed deadline constraints once every minute ($\tau = 60$ seconds).
- *MAUI-DSP.* We implement the MAUI [32] optimization model with *lp_solve* [11] as an enhanced baseline capable of leveraging the DSP locally in the following manner. All pipeline methods that can be executed remotely are flagged as remote-able, and the solver runs its local-remote binary partitioning logic with respect to the DSP, i.e. it will always prefer the low-power DSP over the CPU to compute sensor tasks locally. The solver logic is written in the AMPL modeling language [3] and runs as a service in cloud, while the mobile device is intended to communicate schedules by sending input parameters (sensing tasks identifiers, resource availability) as a JSON string. MAUI annotations are not explicitly implemented, and neither is runtime profiling of the methods. Instead, the solver leverages domain-specific information such as the type of pipeline methods, their expected runtime and energy characteristics from the offline profiling.

7.2 LEO's Overhead

Runtime. The runtime of the scheduling algorithm solving the optimization problem on the DSP is largely determined by the number of sensing apps to be managed. In Figure 6 we plot the runtime of the scheduler as a function of two parameters: the number of managed sensor apps and the rescheduling interval. The runtime for scheduling the execution of 5 apps every second is 107 milliseconds which allows frequently revising the offloading decisions at the expense of a relatively small added latency. In addition, if the algorithm runs once every second when there are generated sensor tasks (triggered by the presence of relevant events such as speech or motion), the daily energy budget for the scheduling on a 2300mAh battery would amount to $< 0.5\%$. We can attribute the success of this relatively low overhead to two factors. First, although the optimization search space grows exponentially in the number of apps,

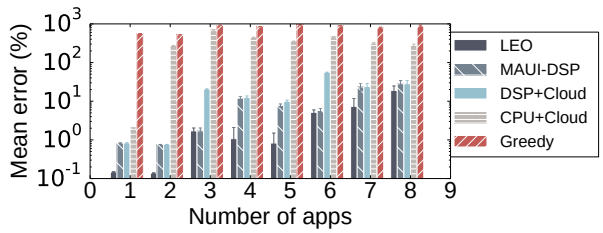


Figure 7: Deviation from the energy of the optimal solution when the offloading variants generate a schedule.

the low latency is enabled by the heuristic algorithm scaling polynomially instead of exponentially with the number of apps. Second, the low energy is maintained by running the scheduler entirely on the DSP and independently from the energy-hungry CPU.

Scheduling Capacity. The total response time of LEO is 2x the rescheduling interval plus the time needed to produce a schedule (which amounts to $\approx$ 2.1 seconds). In other words, the scheduler provides soft real-time guarantees which are sufficient for notification-style sensing applications (e.g., mute the device when entering a conversation, or trigger services based on voice commands). Whereas the scheduling algorithm can solve optimization problems within 500ms for 15 apps on the DSP, typically only a small proportion of the apps will be executed on the DSP as it becomes easily overwhelmed. With 2 hardware threads reserved for application-specific logic, the DSP is able to process with real-time guarantees the feature extraction stages of several apps of the complexity of our examples. Longer-running classification stages need to be broken down into subcomputations, in which case the DSP could typically process a subset of these additional tasks from this stage for one to two more apps. This break-down is achieved through the Pipeline Partitioning discussed in §5.1.

DSP Memory. The runtime memory limit on the DSP of the Qualcomm Snapdragon 800 MDP [20] is currently 8MB of which we use 2MB for system parameters and application models (including 1 DNN and 5 emotion or speaker GMMs). If the LPU Sensing Scheduler revises the joint app schedule every 30 seconds, we would also need approximately 480KB of memory to buffer raw microphone data sampled at a rate of 8KHz. We recall that we use the buffering to monitor the exact workload generated by the currently activated sensor applications. The rest of the memory is reserved to useful application data such as accumulated inferences.

7.3 Optimality of Offloading Schedules

Here we investigate how close our scheduling heuristics as well as straw-man offloading variants are to an optimal schedule. We generate example sensing tasks to be offloaded and compare the generated solutions to the optimal ones produced by an optimization solver. We create sensing traces with a workload that matches 30 seconds of relevant sensing context (detected motion and speech) and vary the number of applications to be scheduled in each trace. For each number of applications we create 10 different traces (where applicable) by changing the underlying set of sensor apps. Application sets are sampled with repeats from the family of implemented example apps shown in Table 3. The generated example configurations are expressed as mixed integer linear programming (MILP) problems via the AMPL modeling language [3] and fed to the optimization solver GLPK [7]. We observe significant, on the order of minutes or more, delays in the termination of the solver when the number of scheduled applications exceeds 8, which is why we limit this number when presenting the results.

In Figure 7 we plot how far off percentage-wise the offloading solutions are from the global optimum found by the GLPK solver. The results show that *LEO produces generally good solutions that are on average within* 5% *away from the optimal ones*. In contrast, the closest among the alternatives, DSP+Cloud and MAUI-DSP, are on average 19% and 10% away from the optimum respectively. As expected, LEO's (and the alternative's) error increases with the rise in number of scheduled apps to reach 19% when 8 apps are scheduled. Nevertheless, we believe that the LPU Sensing Scheduler provides a much needed optimality trade-off to make the offloading decisions practical and executable on the DSP.

7.4 LEO vs Alternatives

In this subseciton we compare the performance of LEO in terms of energy consumption against the commonly found offloading baselines defined in §7.1.

Experimental Setup. We introduce three example scenarios covering the spectrum from light to heavy sensing application usage. Depending on the scenario, the user has subscribed for the services of a different subset of the applications shown in Table 5. To maintain a mixture of applications with a variety of deadlines, we coarsely split the apps into two groups in the following way. We impose near real-time requirements for the accelerometer-based inference algorithms as well as the Keyword Spotting, Speaker Counting and Stress Detection applications by setting their inference deadlines to be equal to their processing period, and set the heavier Emotion Recognition and Speaker Identification pipelines to be tolerant to larger 10-second delays in obtaining an inference (double their period). The delay tolerance for these sensor apps is set as an example to diversify the timeliness requirements.

We generate 100 1-minute long sensor traces per scenario with relevant events sampled from a uniform random distribution. Such events are detected speech and motion that trigger the generation of sensor jobs. We note that even though the length of the sensing trace appears relatively short, it is sufficiently long to enable the sensor apps to generate a large amount of jobs. An Emotion Recognition app, for instance, will create 12 jobs per minute given continuous speech and features are extracted for classification every 5 seconds (Table 3), whereas the Keyword Spotting app would produce 60 jobs per minute. The saturation of sensing context (speech, motion) that generates pipeline tasks varies from 5% to 100% in the artificial traces. For instance, a trace that is 1 minute long might contain 20 seconds of speech (33%) spread throughout the whole interval and grouped into several patches of continuous speech. We replay the traces for each offloading strategy and evaluate the energy consumption depending on the produced distribution of computations among offloading resources and CPU.

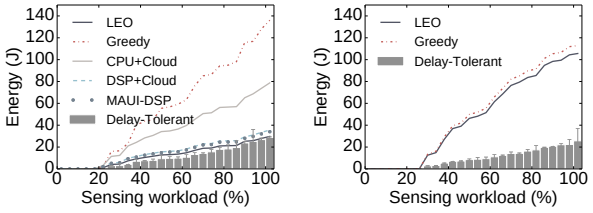
System Load and Energy Profiling. Power measurements are obtained with a Monsoon Power Monitor [14] attached to the MDP. The average base power of maintaining a wake lock on the CPU with a screen and WiFi off is 295mW for the MDP. Each application is profiled separately for energy consumption by averaging power over 10 runs on the CPU, DSP and GPU where applicable. To obtain the power contributed by the sensor processing algorithms only, we subtract the base power from the total power of running the applications in background mode with a screen off. No other background services are running during the profiling apart from the system processes. We confirm that the total sensing system energy consumption is additive as long as the normalized CPU load on the MDP remains below $\approx$ 80%. Thus, the total energy for a sensing trace is an additive function of the energy expenditure of individual tasks under moderate CPU utilization. As reported by DSP.Ear [37] we confirm that popular mobile apps from various categories that follow a processing pattern different from the sense-transform-classify one rarely push CPU utilization beyond 25%.

	WiFi 5Mbps			WiFi 1Mbps			3G 0.8Mbps			3G 0.4Mbps			No connectivity		
	H	M	L	H	M	L	H	M	L	H	M	L	H	M	L
LEO Delay-Tolerant	0.87	0.86	1.00	0.89	0.70	1.00	0.57	0.50	0.58	0.37	0.33	0.32	0.23	0.21	0.30
Greedy	5.30	4.36	4.74	3.97	3.60	3.19	2.64	2.54	1.90	1.64	1.69	1.02	1.04	1.08	1.00
CPU+Cloud	2.86	2.67	2.91	2.49	2.40	3.02	2.07	2.13	1.77	1.75	1.90	1.67	n/a	n/a	n/a
DSP+Cloud	1.17	1.24	1.00	1.20	1.24	1.00	1.23	1.30	1.00	1.21	1.34	1.00	n/a	n/a	n/a
MAUI-DSP	1.13	1.19	1.00	1.11	1.15	1.00	1.08	1.16	1.00	1.04	1.17	1.00	n/a	n/a	n/a

Table 4: Mean factors showing the amount of energy expended by the baselines relative to LEO. A factor of x means that the offloading alternative expends x times the amount of energy for the same workload-connectivity scenario.

	Heavy (H)	Medium (M)	Light (L)
Activity Recognition	✓	✓	✓
Step Counting	✓	✓	
Speaker Counting	✓		✓
Emotion Recognition	✓		✓
Speaker Identification	✓	✓	
Stress Detection	✓	✓	
Keyword Spotting	✓	✓	

Table 5: Applications used in the workload scenarios.



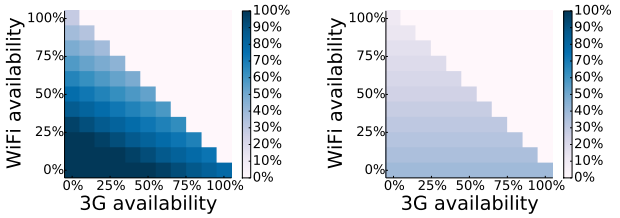
(a) WiFi 5Mbps (b) No connectivity

Figure 8: Energy consumption of the offloading strategies compared against the delay-tolerant LEO as a function of the sensing workload saturation for the medium load scenario (M).

To mitigate any potential interference, we limit the amount of concurrent CPU threads working on sensing tasks to two which keeps the extra CPU load incurred by sensor processing below 40%. If the scheduler decides to use the CPU for computation under high system loads, interference with other services is inevitable unless sensor processing is canceled. In such extreme conditions, delays in the response time of services is expected (e.g., video playback, game play) as well as an increase in the total energy consumption.

Baseline Comparison Results. In Table 4 we display the relative amount of energy incurred by the offloading strategies when compared to LEO. The numbers show how many times the total energy of LEO an offloading alternative consumes. *In all of the resource availability and workload scenarios LEO succeeds in delivering better energy profiles than the alternatives.* The cloud-based baselines, for example, that always perform the classifications remotely and *do not perform cross-app resource optimization*, fail to spot optimization opportunities where processing the classification stages (deeper into the pipeline) on the DSP may be cheaper than remote computations. As a result, the CPU+Cloud strategy consistently consumes 1.6x to 3x more energy than LEO, whereas the DSP+Cloud alternative introduces significant 17% to 34% overheads under heavy and medium workloads. Compared to CPU+Cloud, the DSP+Cloud baseline reduces energy consumption by $\approx$ 2x times on average across the workloads, which is a significant improvement. This energy reduction can be thought of as the gains of simply leveraging an LPU without benefiting from any cross-resource optimization. With principled scheduling of sensor tasks, as we have already shown, energy gains can be up to 34% higher with LEO than with DSP+Cloud for medium to heavy workloads.

In Figure 8 we plot the mean energy expended by the sensing pipelines when following the various offloading schedules as a function of the sensing workload (e.g. proportion of speech captured by the microphone) for the sensing traces. Under relatively good WiFi throughput (5Mbps) LEO consumes 30J for performing all sensor tasks when there is 100% saturation of the sensing context (continuous speech/walking) in the trace. To put this number into perspective and assuming 4.5 hours of talking on average during the day [47], LEO would drain 26% of the capacity of a standard 2300mAh battery (such as the one of a Nexus 5 [9] based on the Snapdragon 800 SoC) to fully process the sensor data for the above mentioned set of accelerometer and microphone applications while importantly maintaining timeliness guarantees. The best among alternatives MAUI-DSP scheduler with its $\approx$ 20% energy overhead would drain the notably higher 31% of the battery to process the same amount of sensor data.



(a) Medium (b) Light

Figure 9: Percentage of the battery capacity needed for processing 4.5 hours of sensing context under varying network availability.

Why not MAUI? A general-purpose offloader such as MAUI-DSP significantly outperforms naïve cloud offloading alternatives, yet there are multiple workload scenarios where LEO can maximize the energy gains even further. For example, under medium and heavy loads LEO can be up to 19% more efficient compared to the version of MAUI enhanced with DSP processing capabilities. This improvement can be attributed to two factors: 1) MAUI's local-remote binary partitioning model does not explicitly model the heterogeneity of local resources and may miss more optimal scheduling configurations that involve multiple local processing units; 2) MAUI applies its offloading decisions structurally for the program at the method level, whereas LEO exploits algorithm semantics to allow copies of the same method with different data (e.g., speaker GMM probability estimation) to be run in parallel on different processing units and cloud. In addition, the original MAUI uses the network to communicate schedules with a remote server to save energy, but still network transfers are mediated through the CPU in a high-power active state (hundreds of mW). Compared to our LPU Sensing Scheduler running on the DSP, MAUI would require about 7 to 10 times more energy for the schedule computation given a rescheduling frequency of 1 second.

Why not Wishbone? Wishbone's [56] original problem formulation targets a slightly different goal: increase data rate and reduce network bandwidth. Minimizing network bandwidth is not a good proxy for energy because recent mobile devices boast powerful multi-core CPUs that can easily cope with complex process-

ing locally – this will incur a high energy cost but will minimize the amount of transferred data. To target on-device energy consumption, the objective needs to be redefined as we have done and latency constraints need to be added to ensure apps remain responsive. Further, frequently using an optimization solver such as GLPK [7] incurs a large energy and computation overhead. We find that scheduling the tasks of no more than 7 or 8 sensor apps such as the examples we have implemented requires on average 100ms on the Snapdragon CPU which is fast enough but the energy cost of running the scheduler there is high – more than 30 times higher than what LEO requires. Given this overhead the only alternative is to run Wishbone less frequently: the solver would need to schedule defensively the execution across resources for multiple apps. In our experiments this leads to missed opportunities of using resources and energy consumption that is more than 3 times higher than what LEO can deliver.

7.5 Practicality Considerations

Varying Network Availability. In Figure 9 we plot the percentage of the battery needed by the system to fully process the sensor data for a sensing workload of 4.5 hours spent in conversations [47] under the medium and light application scenarios (Table 5) and as a function of the network availability. The consumed energy is mindful of the sampling of the sensors and the overhead of waking up the CPU. We vary the amount of time when the system would be able to offload part of the classifications via WiFi or 3G. The network throughput is set to 5Mbps for WiFi and 0.4Mbps for 3G (median uplink throughput for 3G is dependent on carrier but remains around 0.3-0.4Mbps [41]). According to a recent study of smartphone usage patterns of more than 15K mobile users [65], 50% of them are connected to WiFi, LTE or 3G for at least 80% of the time. Being able to offload processing assuming such cumulative wireless coverage in Figure 9 corresponds to draining around 67% of a 2300mAh battery for medium workloads and barely 27% for light scenarios. The figures for the medium workload are high but we stress we maintain near real-time responsiveness for most of the applications. *Should we relax the deadline constraints to use Delay-Tolerant LEO, we can drop these numbers to merely* 25% *and* 12% *for the medium and light scenarios respectively*.

Smartphone Usage. To understand how the workloads in Table 5 affect the user experience we analyze a dataset of 1320 smartphone users, provided to us by the authors of AppJoy [68], where interactive mobile application usage is logged on an hourly basis. We replay the user daily traces with LEO running in the background and add to all traces the workload of typical background services (a mail client, Facebook, Twitter, music playback) in a manner similar to [37]. Assuming the previously mentioned 80% wireless network coverage and 4.5 hours of speech on average in a day (as found by SocioPhone [47]), we find that with the Delay-Tolerant version of LEO *for more than* 80% *or* 93% *of the daily usage instances the users would be able to sustain a full* 24*-hour day of operation without recharging a* 2300*mAh battery when the sensing applications from the medium and light scenarios are active respectively*.

7.6 GPU Acceleration

In this subsection we investigate the implications of scheduling computations when an additional massively parallel heterogeneous processor such as the Qualcomm Adreno 330 GPU [2] is added to the pool of resources available to LEO. We build two scheduling alternatives to streamline our analysis: 1) *LEO-GPU* which follows LEO's scheduling logic and brings the GPU as an optional resource for the two heaviest classification algorithms (GMMs and DNNs discussed in §6); 2) *DSP+GPU* which always uses the GPU for the algorithms that can be executed there, extracts features on the DSP and any routines that cannot meet the deadlines on either of these processors are run on the CPU.

	WiFi		3G		
	5Mbps	1Mbps	0.8Mbps	0.4Mbps	local
LEO-GPU	1.00	1.00	0.74	0.53	0.38
DSP+GPU	2.08	1.55	1.03	0.64	0.46
LEO-GPU (5s)	1.00	0.76	0.49	0.29	0.22
DSP+GPU (5s)	1.13	0.80	0.51	0.31	0.25

Table 6: Mean factors showing the amount of energy expended by the alternatives relative to LEO. The bracketed names refer to the same scheduling strategies when the rescheduling interval is set to 5 seconds which relaxes the deadline constraints for real-time apps and promotes batched GPU execution.

In Table 6 we show the proportion of LEO's energy the GPU-enhanced alternatives would spend in order to process the workloads from Table 5 under varying network connectivity. For each connectivity use case, the numbers are averaged across the heavy, medium, and light scenarios. With slower connections, the GPU-enhanced strategies spend a fraction ($< 0.75x$) of vanilla LEO's energy to process the same workloads which suggests that the GPU is a viable offloading option cheaper than CPU and cloud offloading. With faster connections under tight deadline constraints (rescheduling every second by default), LEO-GPU spends the same amount of energy as LEO which means that the GPU is not used in these faster connectivity scenarios. In our experiments the GPU can deliver results faster than 5Mbps cloud ($\approx 6x$ for the Keyword Spotting and $\approx 3x$ for the Speaker/Emotion Recognition) but consumes more power which is dominated by the GPU initialization stage repeated every second. Interestingly, if we pay the setup costs once and batch multiple computations for GPU execution, LEO-GPU (5s) begins to find opportunities where the total GPU energy is lower than 1Mbps cloud offloading. In other words, LEO-GPU automatically discovers we can compensate for the initially consumed high power with sheer GPU speed.

8. DISCUSSION

We now examine key issues related to LEO's design.

Beyond the DSP. LEO is extensible beyond the three computation classes in our prototype to $n-$units by profiling each supported sensor algorithm under each new processor. As we have shown in §7.6, support for a GPU processor can be easily incorporated into LEO's resource pool modeling but may require extra programmer effort. We anticipate future LEO versions will provide a more comprehensive GPU support and fan-out feature extraction to multiple DSP varieties.

Extending Sensor Support. We largely focus on the accelerometer and microphone sensors as examples of a low and a high-data rate sensors, respectively. As these sensors provide a large variability in their requirements, they are an effective combination to understand the performance of LEO. However, our design is generic enough to support other phone sensors.

Device Fragmentation. Despite interacting with a co-processor, LEO remains portable to various phone models: LEO's components are OS-specific rather than device-specific, with two exceptions. First, each DSP variety needs a runtime and sensor algorithm library. Scaling LEO to use multiple DSPs would require adding support in the scheduling service for communication across different computational units and providing compatible implementations for the sensor algorithms. However, units such as the DSP in the Qualcomm 800 SoC is in dozens of smartphones, and recent DSP programmability trends revolve around adopting standard em-

bedded programming tools such as C. Second, kernel drivers are needed to interface non-CPU components to the OS. But drivers are required only for each {OS, component} combination.

Programmability. We have provided Java wrapper functionality which allows developers to specify custom chains of sensor processing with a few lines of code when the library of pre-built algorithmic components are used. We acknowledge this may not always be possible, in which case the developers can integrate custom algorithms by providing DSP compatible C routines that conform to a set of conventions (briefly outlined in §6) most of which are set by the Qualcomm Hexagon SDK we used in our prototype.

Custom algorithms that do not conform with LEO's partitioning conventions will not benefit as much from the scheduler as structured algorithms. As long as the runtime of these custom algorithms is within the rescheduling interval, LEO will be able to find energy reduction opportunities for concurrent sensing apps without compromising the performance of the introduced new algorithms. This is because the algorithm execution can be treated as a single computational unit that involves the full pipeline (without exposing finer implementation details). When the custom algorithms are long running (severely exceeding the rescheduling interval), and given that the scheduler is not pre-emptive, there might be suboptimal resource utilization choices in light of unforeseen future resource availability. However, such cases are expected to be rare since mobile sensor processing [51, 61, 50, 53, 66, 52] is typically periodic over the sensor stream with short repeated tasks to maintain tight mobile resource consumption and timeliness guarantees.

Proprietary Sensor Processing. Exposing an app's sequence of sensor processing steps to LEO entails intellectual property risks, but this is a problem relevant to a class of emerging orchestrators that operate with domain-specific signal processing knowledge [42, 44, 54]. As these solutions mature, new approaches will be developed to handle security risks. If developers trust the OS, sandboxing techniques [67] can be applied to prevent LEO from leaking sensitive information such as parameters for the classification models. If customized sensor processing C or OpenCL routines need to be added, code obfuscation techniques can be taken advantage of.

9. RELATED WORK

SpeakerSense [50], Little Rock [58], DSP.Ear [37] and AudioDAQ [64] utilize low-power co-processors or purpose-built peripheral devices to achieve energy savings while supporting a fixed set of constantly running sensing applications. However, none of the above mentioned are designed to dynamically balance the workload when the set of actively running sensor apps changes. DSP.Ear's optimizations, for example, alleviate the burden on the memory-constraint DSP by sacrificing inference accuracy. While these techniques are applicable to our set of applications and may complement our system, we focus on handling sensor workloads without modifying the app accuracy.

Why not general-purpose offloaders? General-purpose offloaders [32, 59, 62, 69, 30, 60, 38], do not target the diverse sensor processing workloads explicitly and, as a result, may miss substantial optimization opportunities. For example, MAUI [32] defines an offloading optimization problem that performs binary local-remote split decisions (CPU vs. cloud) and inspects the general program structure but does not take advantage of domain-specific signal processing knowledge. As we have shown in §7, such knowledge can be leveraged for even greater energy benefits. Odessa [59] instruments individual apps to make offloading decisions for improving makespan and throughput but performs only per-app performance tuning instead of cross-app optimizations when apps share scarce mobile computing resources. Code in the Air [62] assumes that wireless connectivity is plentiful and that cloud offloading is the ultimate solution to reduce energy and improve throughput. With the advent of low-power co-processors these assumptions are seriously challenged: we have demonstrated that optimal offloading configurations for sensor processing workloads are the ones that utilize a combination of all available computational resources. Last, VM migration mechanisms [30, 38] offer performance benefits but are difficult to deploy across architecturally different platforms.

Why not other sensor orchestrators? Existing sensor orchestrator frameworks [44, 42, 43, 49, 52, 45] approach the optimization space from different angles in order to improve sensor processing on resource-constraint mobile devices, and often provide complementary functionality that can be executed side by side with LEO. Reflex [49], for example, eases programmability for LPUs [49] but does not explicitly optimize for energy efficiency. MobileHub [63] automatically rewrites app binaries to provide a sensor data notification filter that buffers data on the LPU unlikely to result in an application notification and thus trigger pipeline processing. However, when the later stages of sensor processing pipelines are triggered and execution cannot be bypassed, applications will benefit from LEO automatically distributing chunks of these later-phase computations across resources. CAreDroid [34], on the other hand, presents a framework for automatically choosing among different implementations of the same sensor processing logic that leads to highest performance gains given the current device and user context. Again, we argue that once the relevant processing for an application is determined, sensor computations can be further optimized by *jointly* deciding for the currently active sensor apps on which resource their execution logic should be run. ORBIT [54] similarly to LEO uses profile-based partitioning of application logic to determine the most appropriate resource to use for a processing task issued by a data-intensive embedded app, but does not focus its optimization on multiple simultaneously running apps.

Orchestrator [44] does not scale well with the increase in number of offloading configurations as it systematically explores subsets of offloading plans the number of which grows exponentially with offloading components and sensor apps. Wishbone [56] is very closely related to our work and we build upon some of its fundamentals (linear programming formal model, exploiting data flow semantics for the partitioning). As we have demonstrated in §7, it was originally designed to maximize a different optimization objective and in the case of frequent rescheduling incurs a high energy overhead. SymPhoney [42] introduces a powerful utility-based model to deal with resource contention of sensor apps locally, whereas we attempt to maximize the efficiency of multiple apps with their original maximum utility across the various resources (LPU, CPU, GPU and cloud).

10. CONCLUSION

We have presented LEO, a mobile sensor inference algorithm scheduler enabling concurrent execution of complex sensor apps while maintaining near real-time responsiveness and maximizing energy efficiency. LEO makes this possible by restructuring and optimally partitioning sensor algorithms (from simultaneously running applications) across heterogeneous computational units, and revising this allocation dynamically at runtime based on fluctuations in device and network resources.

11. ACKNOWLEDGMENTS

This work was supported by Microsoft Research through its PhD Scholarship Program. We thank the anonymous reviewers and our shepherd for their valuable comments and suggestions.

12. REFERENCES

[1] Accupedo Pedometer. http://www.accupedo.com/.

[2] Adreno GPU SDK.

[3] AMPL modeling language. http://ampl.com/.

[4] Android Sensor APIs. http://developer.android.com/guide/topics/sensors/index.html.

[5] Apple Motion Core API. https://developer.apple.com/library/iOs/documentation/CoreMotion/Reference/CoreMotion_Reference/index.html.

[6] Apple Siri. https://www.apple.com/uk/ios/siri/.

[7] (GLPK) GNU Linear Programming Kit. https://www.gnu.org/software/glpk/.

[8] Google Cloud Dataflow. https://cloud.google.com/dataflow/model/programming-model.

[9] Google Nexus 5. https://www.qualcomm.com/products/snapdragon/smartphones/nexus-5-google.

[10] HTK Speech Recognition Toolkit. http://htk.eng.cam.ac.uk/.

[11] lpsolve MILP Solver. http://lpsolve.sourceforge.net/5.5/.

[12] Lumia SensorCore SDK. https://www.nuget.org/packages/LumiaSensorCoreSDK/.

[13] Microsoft Cortana. http://www.windowsphone.com/en-gb/how-to/wp8/cortana/meet-cortana.

[14] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[15] Moovit. http://www.moovitapp.com/.

[16] Moves Activity Diary. https://www.moves-app.com/.

[17] OpenCL.

[18] Qualcomm Hexagon DSP. https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor.

[19] Qualcomm Hexagon SDK. https://developer.qualcomm.com/mobile-development/maximize-hardware/multimedia-optimization-hexagon-sdk.

[20] Qualcomm Snapdragon 800 MDP. https://www.qualcomm.com/documents/snapdragon-800-processor-product-brief.

[21] Qualcomm Snapdragon 800 Processors. http://www.qualcomm.com/snapdragon/processors/800.

[22] RunKeeper. http://runkeeper.com/.

[23] Shake Gesture Library Windows Phone 8. http://code.msdn.microsoft.com/windowsapps/Shake-Gesture-Library-04c82d5f.

[24] Shazam. http://www.shazam.com/.

[25] Snapdragon 800 Smartphones. http://www.qualcomm.com/snapdragon/smartphones/finder.

[26] Waze Social GPS Maps and Traffic. https://www.waze.com/.

[27] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[28] A. Brajdic and R. Harle. Walk detection and step counting on unconstrained smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 225–234, New York, NY, USA, 2013. ACM.

[29] G. Chen, C. Parada, and G. Heigold. Small-footprint keyword spotting using deep neural networks. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'14, 2014.

[30] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[31] C. Cotta and A. J. FernÃąndez. Memetic algorithms in planning, scheduling, and timetabling. In K. P. Dahal, K. C. Tan, and P. I. Cowling, editors, *Evolutionary Scheduling*, volume 49 of *Studies in Computational Intelligence*, pages 1–30. Springer, 2007.

[32] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62. ACM, 2010.

[33] A. de Cheveigné and H. Kawahara. YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[34] S. Elmalaki, L. Wanner, and M. Srivastava. Caredroid: Adaptation framework for android context-aware applications. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 386–399, New York, NY, USA, 2015. ACM.

[35] Z. Fang, Z. Guoliang, and S. Zhanjiang. Comparison of different implementations of mfcc. *J. Comput. Sci. Technol.*, 16(6):582–589, Nov. 2001.

[36] J. Gemmell, G. Bell, and R. Lueder. Mylifebits: a personal database for everything. *Communications of the ACM (CACM)*, 49(1):88–95, January 2006. also as MSR-TR-2006-23.

[37] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. DSP.Ear: leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, New York, NY, USA, 2014. ACM.

[38] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153–166, New York, NY, USA, 2013. ACM.

[39] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *J. Acoust. Soc. Am.*, 57(4):1738–52, Apr. 1990.

[40] G. Hinton, L. Deng, D. Yu, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. S. G. Dahl, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012.

[41] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 165–178, New York, NY, USA, 2010. ACM.

[42] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[43] 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 *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[44] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Eigth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2010, March 29 - April 2, 2010, Mannheim, Germany*, pages 135–144, 2010.

[45] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, IPSN '16, pages 23:1–23:12, Piscataway, NJ, USA, 2016. IEEE Press.

[46] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, Sept. 2010.

[47] Y. Lee, C. Min, C. Hwang, J. L. 0001, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song. Sociophone: everyday face-to-face interaction monitoring platform using multi-phone sensor fusion. In H.-H. Chu, P. Huang, R. R. Choudhury, and F. Zhao, editors, *MobiSys*, pages 499–500. ACM, 2013.

[48] R. LiKamWa, Y. Liu, N. D. Lane, and L. Zhong. Moodscope: Building a mood sensor from smartphone usage patterns. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 389–402, New York, NY, USA, 2013. ACM.

[49] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 13–24, New York, NY, USA, 2012. ACM.

[50] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu. Speakersense: Energy efficient unobtrusive speaker identification on mobile phones. In *Proceedings of the 9th International Conference on Pervasive Computing*, Pervasive'11, pages 188–205, Berlin, Heidelberg, 2011. Springer-Verlag.

[51] H. Lu, D. Frauendorfer, M. Rabbi, M. S. Mast, G. T. Chittaranjan, A. T. Campbell, D. Gatica-Perez, and T. Choudhury. Stresssense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 351–360, New York, NY, USA, 2012. ACM.

[52] 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 *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 71–84, New York, NY, USA, 2010. ACM.

[53] C. Luo and M. C. Chan. Socialweaver: Collaborative inference of human conversation networks using smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 20:1–20:14, New York, NY, USA, 2013. ACM.

[54] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing. Orbit: A smartphone-based platform for data-intensive embedded sensing applications. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 83–94, New York, NY, USA, 2015. ACM.

[55] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P Report 826, California Institute of Technology, 1989.

[56] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: ProoĭňĄle-based Partitioning for Sensornet Applications. In *NSDI 2009*, Boston, MA, April 2009.

[57] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 403–416, New York, NY, USA, 2013. ACM.

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

[59] M. Ra, A. Sheth, L. B. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*, pages 43–56, 2011.

[60] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, pages 73–84, New York, NY, USA, 2011. ACM.

[61] 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 *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, Ubicomp '10, pages 281–290, New York, NY, USA, 2010. ACM.

[62] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM.

[63] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 227–238, New York, NY, USA, 2015. ACM.

[64] S. Verma, A. Robinson, and P. Dutta. Audiodaq: Turning the mobile phone's ubiquitous headset port into a universal data acquisition interface. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 197–210, New York, NY, USA, 2012. ACM.

[65] D. Wagner, A. Rice, and A. Beresford. Device analyzer: Understanding smartphone usage. In *10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2013.

[66] C. Xu, S. Li, G. Liu, Y. Zhang, E. Miluzzo, Y.-F. Chen, J. Li, and B. Firner. Crowd++: Unsupervised speaker count with smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 43–52, New York, NY, USA, 2013. ACM.

[67] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, Bellevue, WA, 2012. USENIX.

[68] B. Yan and G. Chen. Appjoy: Personalized mobile application discovery. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*.

[69] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, Oct. 2014. USENIX Association.

[70] G. Zhou, J. H. L. Hansen, and J. F. Kaiser. Nonlinear feature based classification of speech under stress. *IEEE Transactions on Speech and Audio Processing*, 9(3):201–216, 2001.