The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing

Aman Kansal Scott Saponas A.J. Brush Kathryn S. McKinley Todd Mytkowicz
Ryder Ziola

Microsoft Research
kansal@microsoft.com

Abstract

Emerging mobile applications that sense context are poised to delight and entertain us with timely news and events, health tracking, and social connections. Unfortunately, sensing algorithms quickly drain the phone's battery. Developers can overcome battery drain by carefully optimizing context sensing but that makes programming with context arduous and ties applications to current sensing hardware. These types of applications embody a twist on the classic tension between programmer productivity and performance due to their combination of requirements.

This paper identifies the *latency, accuracy, battery* (LAB) abstraction to resolve this tension. We implement and evaluate LAB in a system called Senergy. Developers specify their LAB requirements independent of inference algorithms and sensors. Senergy delivers energy efficient context while meeting the requirements and adapts as hardware changes. We demonstrate LAB's expressiveness by uising it to implement 22 context sensing algorithms for four types of context (location, driving, walking, and stationary) and six diverse applications. To demonstrate LAB's energy optimizations, we show often an order of magnitude improvements in energy efficiency on applications compared to prior approaches. This relatively simple, priority based API, may serve as a blueprint for future API design in an increasingly complex design space that must tradeoff latency, accuracy, and efficiency to meet application needs and attain portability across evolving, sensor-rich, heterogeneous, and power constrained hardware.

1. Introduction

Emerging personalized mobile applications and operating system services based on user *context* have the potential to revolutionize the mobile experience, but only if they do not render your phone powerless. Mobile context sensing can infer your location [24], movements [14], social situation [20], mood [28], and stress levels [27]. Applications [16] and operating system services [39] can sense context in the background and act when the user transitions into a relevant context. For example on iOS and Android, applications can request a callback when the user reaches a specified location using OS APIs.

Unfortunately, continuously computing context drains mobile batteries quickly. We found Android's `addProximityAlert` API on HTC Desire S reduces battery standby time from 430 to 12 hours. Since mobile devices do more than sense location and users expect at least 16 hours of battery life [1], context sensing is not yet energy efficient enough for continuous use.

Context sensing can however be made energy efficient. For example, if an application can tolerate sensing every one minute instead of continuously, the standby battery life on the Android HTC Desire S doubles to 24 hours. Optimizations such as using the accelerometer for a few seconds every minute to detect user movement and then trigger location sensing only if the user is mobile further increases battery life by 300%, to over two days. In fact, prior research proposed a range of energy efficient continuous context sensing algorithms [4, 7, 14, 18, 19, 24, 38, 40].

The problem is that not all available techniques benefit every scenario. The application developer must choose the appropriate algorithm and its parameters. Developers could potentially implement and characterize these algorithms and determine which one is most efficient for their application, but this approach increases the programming burden and some developers lack the resources, time, or expertise. Even when a motivated developer implements the right sensing algorithm, it only serves one application and other applications

may continue to drain the battery performing the same task. To make context sensing both efficient and widely useful to programmers requires an appropriate abstraction.

This paper identifies, implements, and evaluates the Latency, Accuracy, and Battery life (LAB) abstraction, which seeks to simultaneously obtain programmer productivity and efficiency for continuous context sensing. Applications convey priorities and requirements on the *latency* at which a context change is detected, the *accuracy* of the inferred context, and *battery* consumed. The implementation of the abstraction selects and tunes its context sensing algorithms to meet the constraints and optimizes battery life by adjusting sensing frequency, modality, complexity of signal processing, communication with cloud services, etc. This abstraction simply expresses the tradeoffs between latency, accuracy, and battery life.

We implement and evaluate this abstraction in a prototype continuous context sensing OS service, named Senergy. We implement 22 context algorithms and six example applications. While a user study of programmer productivity is beyond the scope of this paper, we do show that the LAB abstraction captures a wide variety of application requirements and simplifies context programming. We further show that Senergy efficiently satisfies application constraints by choosing among the multiple algorithms and that Senergy may optimize for multiple simultaneous applications.

This work makes the following contributions.

1. We identify and propose the LAB context sensing abstraction in which applications specify latency, accuracy, and battery life priorities and requirements, independent of any particular context algorithm. We implement this abstraction in Senergy and demonstrate how applications use the Senergy API in simple and sophisticated ways.
2. We characterize the tradeoff in latency, accuracy, and energy offered by 22 location and activity (driving, walking, stationary) context inference algorithms. Based on the quantified tradeoffs, we describe how Senergy selects the most appropriate context-inference approach at runtime to simultaneously satisfy application requirements whenever possible and optimize resource use within and across applications.
3. We evaluate Senergy using six applications with varying latency and accuracy requirements. The results, using over 4,200 usage hours of real-world data traces collected from 49 participants, show that Senergy reduces energy use, often by an order of magnitude, while satisfying application requirements.

As the complexity of hardware and software continues to explode, operating system APIs will carry a much higher burden. In particular, emerging mobile applications are using sensors, search, history, and inference to give you ever richer personalized services while at the same time the sensor-rich mobile hardware offers substantially better and new features with every generation. We believe that the priorities and requirements specification in Senergy's API is a promising blueprint for managing this complexity.

2. Motivation

Current systems for continuously sensing context achieve either programmability or energy efficiency, but not both. To achieve both goals requires a new *abstraction*. A good programming abstraction succinctly expresses functionality, such that developers convey *what* they need, but not *how* to do it. The application programming interface (API) correctly and efficiently implements the abstraction. For instance, the `MapReduce` API expresses a parallel data processing abstraction [8]. Developers specify *what* tasks are parallel and the runtime determines *how* to parallelize those tasks.

For continuous location and activity context, we propose the LAB abstraction which expresses: *latency*: how long the system takes to detect a context change; *accuracy*: correctness of the detected change; and *battery*: what fraction of the battery context sensing may consume over a day. These properties ensure application developers express *what* they need from context sensing, but they do not overly constrain *how* the implementation meets these requirements.

We show that this abstraction is sufficient to communicate a wide variety of application requirements for continuous context. Furthermore, we show that the system has sufficient flexibility to simultaneously meet these requirements when possible, optimize for efficiency, *and* optimize across multiple applications. Section 3 discusses how we implement this abstraction in the Senergy API.

However, attaining efficiency remains a challenging problem in its own right. Selecting the best algorithms requires analyzing latency, accuracy, and battery on a myriad of mobile devices and user behaviors. Consider Figure 1, which shows the energy latency tradeoff for six continuous location tracking algorithms that we implement. Variability in latency and energy (x and y axes, respectively) makes the choice of

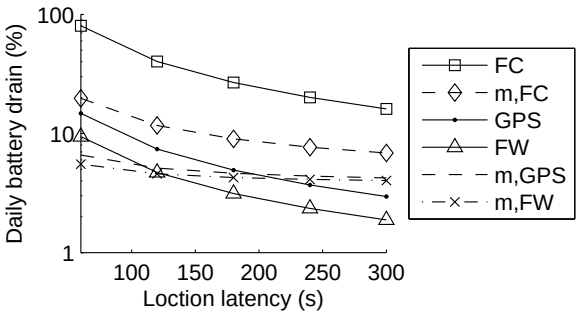


Figure 1: *Location Tracking. Latency-energy tradeoffs for six algorithms averaged over four devices.* The GPS algorithm only uses GPS data, FC uses network Fingerprints over a Cellular data connection, and FW uses Fingerprints over a WiFi connection. The $m, *$ versions sense location only if they first detect movement using accelerometer data.

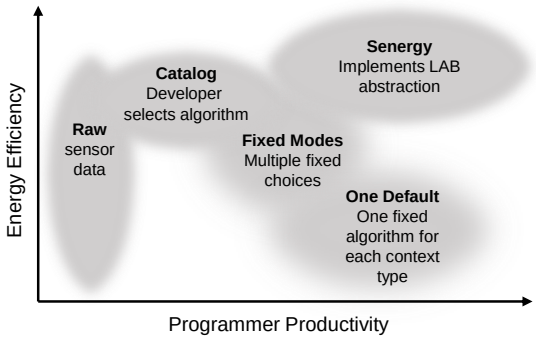


Figure 2: Efficiency and productivity tradeoff for possible context APIs.

algorithm complex. The best algorithm for a particular application depends on its latency requirements, available sensors, and (not shown) user behavior.

For example, consider two location applications. *BuyGroceries* notifies my spouse with a text message if I stop at a grocery store for over 5 minutes, asking for any additions to our list. *WalkHome* texts parents the location of middle school students when school ends and then updates it every minute until they arrive home. Due to their different latency requirements (300 and 60 seconds), Figure 1 shows that different algorithms are optimal. Sections 4-7 discusses how Senergy uses existing context sensing algorithms to meet application specifications and deliver efficiency.

3. Senergy API Design

The Senergy API seeks to give mobile-application developers an intuitive and stable API for context that maximizes programming productivity and to give API developers the opportunity to optimize energy. We describe the limitations of existing and other potential API design alternatives and then describe how Senergy addresses these limitations.

3.1 Context Sensing API Design Space

Figure 2 depicts the programmability and efficiency of the range of choices in the context sensing API design space.

Raw: The OS makes raw sensor data available to the developer. Android, iOS, and Windows Phone provide this API. Developers must implement their own context algorithms using the raw data. As Figure 2 highlights, in theory this approach can be very energy efficient since applications can perform the minimum sensing required for their needs. In practice, it is quite difficult for developer to optimize energy efficiency because they have limited access to energy measurements across multiple devices and user behaviors. Multiple applications that need the same context and run on the same device will duplicate functionality and waste energy. On the programmability axis, this approach is the most burdensome for developers.

One Default: The OS implements one default sensing algorithm for each context, tuned to some expected application use. The Android and iOS platforms provide such a default for location proximity. A single default has the potential to improve programmer productivity. It can also be energy efficient if the context sensing algorithm matches the needs of applications. Unfortunately, as shown in Figure 1, location tracking algorithms have a wide range of latency and energy characteristics. For instance, the Android API `addProximityAlert` is tuned for low latency and for applications that do not need low latency, this default wastes energy. If Android changes the default settings to reduce energy usage, it may break applications tested with the current low latency setting.

Fixed Predefined Modes: Providing multiple fixed modes could overcome the limitations of one default. For instance, the OS may offer three modes: "energy efficient", "low latency", and "high accuracy." Fixed Modes fall between Raw and Default in terms of productivity because developers do not have to implement their own algorithms, but they do need to understand and choose among the modes. Fixed Modes improve energy efficiency over Default because one of the modes is likely closer to the application requirements than the single default.

The disadvantage of Fixed Modes is that they still close off too much of the design space. For example, suppose an application needs high accuracy and can tolerate high latency, but no mode exposes that choice. Fixed modes also shoehorn the OS into rigid contracts and limit extensibility. If a new algorithm emerges that dramatically improves energy efficiency but slightly violates the latency contract in the existing energy efficient mode, the OS cannot use it without introducing a new mode or violating the contract.

Catalog: The OS could expose all its algorithms to applications. A developer then chooses an algorithm and sets its parameters. This type of API can achieve high efficiency since developers may select the most appropriate algorithm and settings for their purpose. Catalog improves programmer productivity over Raw because a developer need not implement algorithms from scratch. However, developers must still determine the best choice and parameter settings for their applications. Choosing is challenging because it may depend on the user's environment, behavior, and device characteristics. If a better context algorithm is added to the Catalog, developers must change their application. If two applications run simultaneously and need the same context, but choose different algorithms, they perform redundant work and waste energy.

3.2 Senergy API Specification

Our goal is an API design that increases both programmer productivity and energy efficiency. We organize our design based on two insights. (1) Specifying a *priority order* among accuracy, latency, and energy use (battery) gives developers a simple way to express many application constraints and gives the implementation flexibility to optimize. (2) A *partial quantitative specification* of latency and/or battery con-

sumption gives developers a more powerful and expressive API, since priorities do not promise specific values. Because accuracy requires ground truth from the user or other source, guaranteeing or quantifying context accuracy at runtime is beyond our scope, and therefore, developers cannot specify a quantitative accuracy value. We show that this LAB (latency, accuracy, battery) abstraction is sufficient for a wide range of context applications.

We implement the LAB abstraction in the Senergy API, which exposes location and activity context through an asynchronous method, `ChangeAlert`. The application registers a callback method, and Senergy invokes it each time Senergy detects the specified change in location or activity. Applications may unsubscribe with `UnsubscribeAlert` .

Table 1 lists the API and arguments. The first method argument specifies the *context*, which consists of *locations* and *activities* in Senergy. We leave other types of context (e.g., mood, attentiveness) to future work. Location and activity by themselves are extremely powerful for many applications including movement tracking, health monitoring, and safe phone interactions while driving. Furthermore, their detection algorithms expose a range of tradeoffs that we use to explore the latency, accuracy, and energy optimization space.

Location elements are geographic coordinates. Activities are values in an enumerated type and currently include *all, driving, walking,* and *stationary*. Since each activity is a parameter, adding new activities is transparent to existing applications. The not operator signifies exiting a location or activity. For instance, `Activity.DRIVING` detects the beginning of driving and ∼`Activity.WALK` detects when the user stops walking. The parameter values `Activity.ALL` and `Location.ALL` result in a callback on all activity and location changes respectively. For instance, `ChangeAlert` (`Location.ALL`) continuously tracks user location.

The second and subsequent arguments are optional. The second argument specifies the highest priority choice of accuracy, latency, or energy. The optional third argument either specifies the next highest priority dimension or quantifies the prioritized dimension for either latency (seconds) or energy (% battery over 24 hr). Subsequent arguments are used similarly. For example, the application may quantify latency at 120 s or battery at 5% of total capacity. Applications cannot quantify accuracy, as discussed above, but including an accuracy priority influences context algorithm selection. For instance, if the highest priority dimension is latency, followed by accuracy, then the available context sensing algorithms are first ranked by latency and then accuracy. If accuracy priority is omitted, Senergy ranks algorithms by battery efficiency instead of accuracy.

Senergy defaults unspecified constraints to the most battery efficient algorithm, subject to OS determined thresholds on useful accuracy and latency. The thresholds are used because a very low energy algorithm that simply hardcodes the location to 'planet earth' is not useful context.

Arguments	**Argument description**
`ChangeAlert`	
Context[]	collection of locations or activities to detect
`ChangeAlert`	
Context[]	collection of locations or activities to detect
FirstPriority	one of accuracy, latency, or battery
`ChangeAlert`	
Context[]	collection of locations or activities to detect
FirstPriority	one of accuracy, latency, or battery
Value	quantitative constraint for first dimension*
`ChangeAlert`	
Context[]	collection of locations or activities to detect
FirstPriority	one of accuracy, latency, or battery
SecondPriority	one of accuracy, latency, or battery
`ChangeAlert`	
...	variable number of optional arguments
`ChangeAlert`	
Context[]	collection of locations or activities to detect
FirstPriority	one of accuracy, latency, or battery
Value	quantitative constraint for first dimension*
SecondPriority	one of accuracy, latency, or battery
Value	quantitative constraint for second dimension*
ThirdPriority	one of accuracy, latency, or battery
Value	quantitative constraint for third dimension*

Table 1: **Senergy API.** All arguments use custom data types, except for constraint values which uses a `double`. *Quantitative constraint is not used for accuracy priority.

Context sensing algorithms meet constraints probabilistically on average across large populations. Sensor, network, user, and device variation may prevent the system from meeting constraints systematically or on occasion. Senergy satisfies as many constraints as possible in priority order.

Applications or the OS may wrap `ChangeAlert` in order to *track* a particular context state over time or distance (e.g., time spent at work or distance moved while in driving); obtain *current* context without initiating continuous sensing; or to expose certain special cases for backward compatibility (e.g., `addProximityAlert` which is simply a call to `ChangeAlert` with a single location as the first parameter).

3.3 Example API Usage

Senergy supports developers with a range of expertise levels, from those who simply use defaults to advanced developers who tune context to match their application needs. We explore this range of programmability with three examples.

Example 1. To detect when the user starts walking without any constraints, an application invokes

```
Activity[] activities = {Activity.WALKING};
ChangeAlert(activities);
```

Senergy uses its default activity algorithm for walking, which provides low battery drain subject to minimum thresholds on accuracy and latency. This usage minimizes programmer effort, and is equivalent to specifying battery as the only priority.

Example 2. To detect each time a user starts driving and count car trips, assuming the application can tolerate a 5 minute latency for the start driving notification, since most driving trips will last at least 5 minutes, it invokes

```
Activity[] acts = {Activity.DRIVING};
ChangeAlert(acts,Priority.Latency,300);
```

Senergy selects an algorithm that detects driving at 300 s latency with high probability. Among all such feasible options it chooses the most energy efficient algorithm that respects a minimum accuracy threshold. This algorithm meets the programmer specification, but it may not provide the most accurate context feasible. We believe that specifying only the critical constraints improves programmer productivity.

Example 3. With the same objective as above, the developer now also wants to restrict the battery impact of the application to 1% over 24 hours, to help ensure users choose to keep it installed. The developer still prefers a 5 minute latency, and would additionally like the highest possible accuracy. The application invokes

```
ChangeAlert(acts,Priority.BATTERY,1,
Priority.Latency,300,Priority.Accuracy);
```

Senergy chooses an algorithm that satisfies as many constraints as possible. If multiple choices meet the 1% battery and 300 s latency constraints, Senergy chooses the one with the highest expected accuracy. If no algorithm achieves 5 minute latency within 1% battery budget, it uses a longer latency, since battery is the highest priority. Specifying all constraints requires extra programmer effort but much less than implementing an algorithm from scratch, or manually selecting an algorithm from a catalog.

4. Senergy Resource Optimization

The API design described above gives the OS significant flexibility for optimizing and evolving context sensing algorithms, without exposing implementation details and forcing unnecessary constraints. We implement Senergy on top of existing sensing capabilities in mobile devices. Figure 3 shows the overall architecture. The hardware abstraction layer (HAL) consists of components that already exist in mobile OSs: raw sensor drivers and a location stack that can obtain location using GPS or network fingerprints (WiFi access points, cellular base station IDs, and their signal strengths), using an Internet service to convert fingerprints to location.

Senergy needs multiple algorithms for each context type since they offer different accuracy, latency, and energy efficiency. At runtime, Senergy selects the most suitable algorithms and parameter values based on application needs. For these purposes, we add three components: two context monitoring components, denoted *Activity Context* and *Location Context*, that include multiple algorithms to continuously sense and infer user context in the background, and the *runtime algorithm selection and tuning* component that implements the runtime logic to select appropriate algorithms and tune their parameters, based on application requirements.

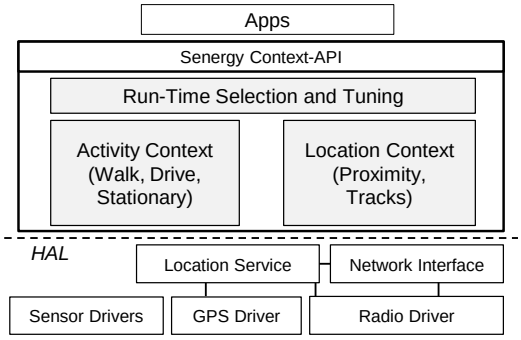


Figure 3: Senergy architecture.

These three components comprise the bulk of the Senergy implementation and are described in the next three sections. Application developers need not understand any of these implementation details to use the API.

Algorithms: A variety of context inference algorithms for movement activity and location monitoring are available [3, 7, 14, 18, 23, 24, 30, 32, 37, 40]. We select representative ones and study the tradeoffs in their energy, accuracy, and latency. We choose *general* algorithms that are as independent as possible of particular user behaviors and their environments. For instance, we do not include methods that require the sensor to be mounted on a particular position on the body [37] or constant WiFi availability, and do not assume that users will park a minimum distance from their destination, work and live in two different locations, etc. Some algorithms exploit non-universal infrastructure where available to improve efficiency, but have fall backs for other environments.

5. Activity Context

We implement (1) three binary classifiers that detect the presence or absence of driving, walking, and stationary, and (2) a multi-state activity classifier that infers if the user is driving, walking, or stationary. For applications that require a single movement state, the binary detectors deliver higher accuracy, while the multi-state classifier is better when all states are to be distinguished.

Activity inference algorithms employ a typical machine learning approach: *sense* over a time window, *compute features* from the sensor data (e.g., features listed in Table 2 for driving), and *classify* the data using a model previously trained on ground truth. In Senergy we use a Naïve Bayes classifier with supervised discretization. We chose this technique for simplicity of implementation and because it generates posterior probabilities that may be used as a confidence measure by applications. We expect other classifiers to also work well (e.g., decision tree, support vector machine).

5.1 Driving Activity

Table 2 lists seven algorithms that we implemented for driving detection, using the accelerometer, cellular, and GPS

Algorithm	Sensor	Features
D-ACC	Accelerometer	magnitude (0.5-3 Hz), magnitude (20-25 Hz), variance
D-GPS	Assisted GPS	speed
D-AG	ACC and GPS	D-ACC and D-GPS
D-CELL	Cellular fingerprint	new towers in window compared to past windows
D-CA	CELL and ACC	D-CELL and D-ACC
D-GC	GPS and CELL	D-GPS and D-CELL
D-AGC	ACC, GPS, CELL	combines all algorithms

Table 2: Driving detection algorithms.

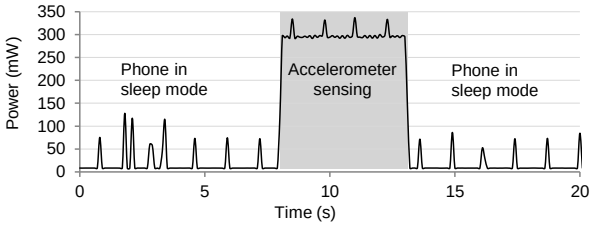


Figure 4: Background accelerometer power

data [23, 30, 32, 37]. Algorithm D-ACC is based on the accelerometer, using features from [30]. D-GPS uses speed sensed using GPS. D-CELL is largely inspired by [32] but differs in that we only consider changes in detected towers (we do not consider features based on signal strength as we did not find them effective in our environment). The remaining algorithms combine multiple sensors to achieve better accuracy or enhance generality. We tuned the feature sets of all the activity algorithms to improve their accuracy based on our multiple user deployments in indoor, outdoor, urban, and suburban environments.

5.2 Driving Activity Tradeoffs

These algorithms provide a design space that Senergy uses to tradeoff accuracy, latency, and energy. We quantify these tradeoffs below to select the best operating point.

We consider continuous background sensing while the device is not in active use, since active time is only a small fraction of the total time for which the device is carried by the user. For background sensing, energy drain includes not just the sensors but also the processing or storage components used to drive the sensors and infer context. Figure 4 shows a sample accelerometer power trace on a Samsung Focus smartphone. We measure power by connecting a power meter to the phone's battery terminals. Power includes the accelerometer sensor, processor, and all active motherboard components when the accelerometer collects data.

We measure energy on the four devices listed in Table 3. Table 4 lists the sensing energy for each sensor used in activity detection algorithms. Our measurements are similar to previous ones on Nokia N95 [24], Android G1 and ATT Tilt [18, 40], and iPhone [35]. All our experimental phones use assisted GPS (a-GPS). With a-GPS, cold and warm start burn about the same amount of energy, and we report a single number.

OS Platform	Devices	
Android OS 2.3.2	HTC Desire	Nexus S
Windows Phone 7.1	Samsung Focus	Asus E600

Table 3: Experimental mobile devices.

Sensor	Avg. Energy (mJ)	Std. Dev. (mJ)
Accelerometer (ACC)	506	70
Assisted GPS (GPS)	2049	159
Cellular fingerprint (CELL)	20	NA

Table 4: Energy use averaged across multiple devices. Network fingerprint energy is measured only on the E600.

The total energy of a context inference algorithm depends on which sensors it uses and how often. For instance, D-CELL collects the cellular fingerprint multiple times to infer if the user is driving, and actual inference energy is higher than the single sample energy listed in Table 4.

Since accuracy, latency and energy characteristics of context algorithms vary significantly with user behavior and environment, we collected several experimental data sets (described in Section 8) across multiple participants in different environments. While none of our participants may exactly match a particular user's behavior or environment, our data provides a statistical estimate of the tradeoffs between algorithms. It would be ideal to learn the tradeoffs individually on every user but computing accuracy requires ground truth labels that are difficult to obtain for every user.

Figures 5a and 5b report the accuracy and latency tradeoffs with energy for the seven algorithms. For three of the algorithms, latencies lower than 600 s are not feasible, truncating those curves. All algorithms have reasonable accuracy, precision, and recall, although the higher energy algorithms do better. Here we also measured *precision* and *recall*, since accuracy alone can be misleading. For instance, when users drive only a small fraction of the time, an algorithm that always outputs not-driving will be accurate. However, this algorithm has poor recall. Accuracy refers to how closely the values returned by the sensing algorithms match the ground truth. Precision is $tp/(tp + fp)$ where tp are true positives and fp are false positives, and recall is $tp/(tp + fn)$, where fn are false negatives. Precision indicates how many of the detected instances were indeed correct, while recall measures how many of all true instances were detected.

Different algorithms may be suitable for different application requirements. For instance, D-CELL minimizes energy if the application can tolerate a few minutes of latency. Prior work detecting driving using cell tower data [21, 32] also reported latencies in minutes. If the application requires

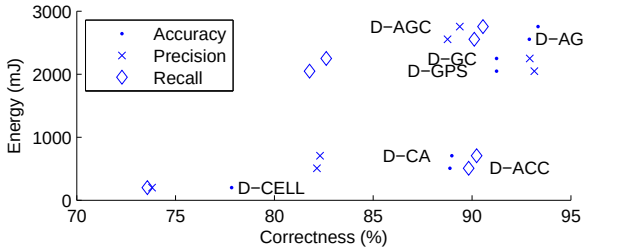
low latency, D-ACC is a good candidate. D-CA yields a small advantage in accuracy at a modest increase in energy but has a much higher latency.

5.3 Walking, Stationary, and All Activities

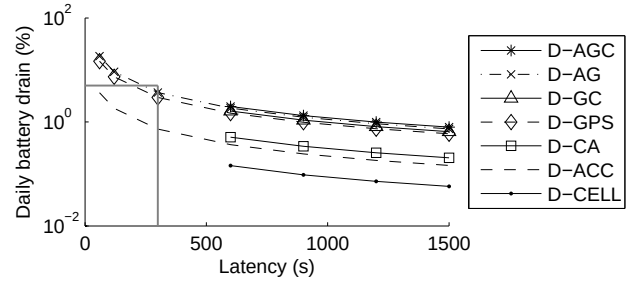
We use the same methodology to implement, measure, and choose the algorithms for walking, stationary, and all activities as described above for driving detection. The classifiers use accelerometer data, GPS data, and their combination. We omit the use of cellular fingerprints for walking and stationary detection, because although feasible in some environments [32], we found that it did not work well in our environment. We implement three binary classification algorithms each for walking (W-ACC, W-GPS, and W-AG), stationary (S-ACC, S-GPS, and S-AG), and multi-state activity (M-ACC, M-GPS, and M-AG).

5.4 Walking, Stationary, and All Tradeoffs

Figures 6a and 6b show the latency and accuracy tradeoffs with energy for walking detection. The algorithms that rely on GPS (W-GPS and W-AG) use significantly more energy than W-ACC, but do not always improve accuracy. W-GPS in fact has lower accuracy than W-ACC because sometimes GPS is not available for long periods such as when the user is indoors. We thus exclude W-GPS from Senergy. W-AG has the potential to improve accuracy but the accuracy difference is smaller than the error bars in our data, suggesting that GPS is not better than the accelerometer for this purpose.



(a) Accuracy



(b) Latency

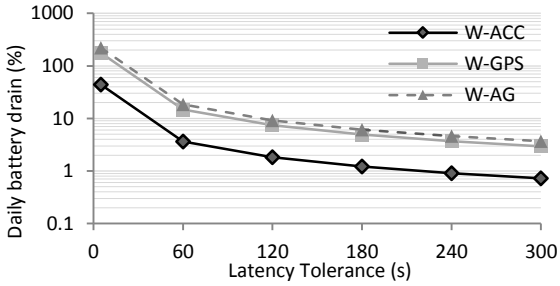
Figure 5: *Driving Detection* accuracy, latency and energy tradeoffs. Accuracy, precision, and recall points at one energy level correspond to the same algorithm label. Standard deviations (not shown for readability) on all users are below 5% except for D-CELL, where they are as high as 15%. Section 7 describes the solid gray lines.

Figures 7 and 8 show the accuracy of stationary context and the multi-class detector. They consume the same amount of energy as walking detection (Figure 6a) because the sensor data collection, featurization, and inference steps are similar. The use of GPS only slightly improves accuracy for these activities. The accuracy for the multi-class detector is lower with larger error bars than the binary detectors, justifying the use of separate binary detectors when possible.

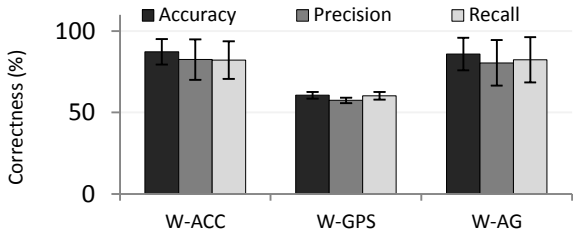
6. Location Context

Applications may invoke the Senergy API with `Location.ALL` to track all location changes or with a set of specific locations to monitor proximity to just that set. We first describe continuous tracking and then introduce additional optimizations when detecting only a specific set of locations.

Continuous Tracking. The literature proposes several techniques to optimize energy for location tracking [3, 7, 14, 15, 18, 24, 40]. We implement one broadly applicable mech-



(a) Latency and Energy



(b) Accuracy

Figure 6: *Walking detection* latency, energy, and accuracy tradeoffs. The highest energy point uses a latency of 5 s. Logarithmic scale is used for the energy axis to capture the wide range of tradeoff points.

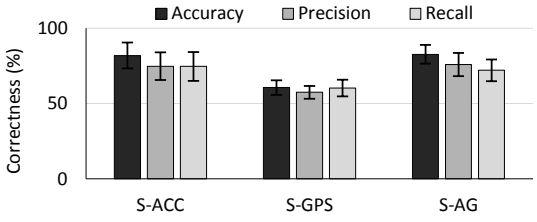


Figure 7: Accuracy for stationary activity.

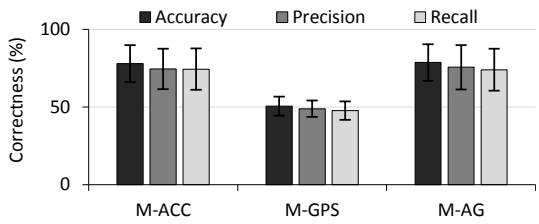


Figure 8: Accuracy for multi-class activity detection.

anism of using low power sensors to detect user movement and sensing location only when movement occurs. [3, 15, 24]. Other mechanisms blacklist regions where GPS is not available or WiFi fingerprints are not useful [40], but we do not implement these because they require extensive learning for the specific environment.

Senergy detects if the user is moving, using the stationary activity detection algorithm, S-ACC. The movement check interval is the smaller of the requested location update latency or two minutes (since the user will typically move for at least that long if their location is to change significantly). If the user is not stationary, location is updated. Two choices are available to update the location, using the GPS or network fingerprints. The key points worth noting are that (1) accuracy varies with the user environment [18] and either network fingerprint or GPS based location can be more accurate depending on whether the user is indoors, within dense WiFi deployments, or not, and (2) the energy use of network fingerprint based location can be higher or lower than GPS (see Tables 4 and 5), depending on whether WiFi (FW) or cellular data connection (FC) is used for Internet access.

We find that once the user starts moving, the previously observed availability or unavailability of GPS satellites or WiFi connectivity for Internet access may change quickly. Since accuracy is not correlated with energy, Senergy cannot tradeoff accuracy for energy. Hence, Senergy first attempts to update location using GPS and if that fails, it uses network fingerprint based location.

Location Proximity. If an application does not require all location changes but only proximity to specific locations, then Senergy performs an additional optimization. We reduce the number of location updates based on the current distance from the nearest interesting location. The intuition is that if the nearest interesting location is a distance x away, and the maximum speed of movement is v_{max}, then it will take the user at least $t_{min} = x/v_{max}$ time to reach the new location. The value of v_{max} could use the maximum speed limit in the country, except when the previous location is near an air strip where maximum aircraft speed may be used. Hence, the next location update may happen after a wait of slightly less than t_{min}. If t_{min} is smaller than the delay tolerated by the application, Senergy will use the tolerable delay. If t_{min} is so large that a location update at that interval is cheaper in energy than periodically sensing movement, the movement sensing is dropped until t_{min} reduces. Whenever Senergy updates location, the algorithm checks if the location is within a specific radius of one or more of interesting locations and notifies the application accordingly. When the number of interesting locations is very large and there is always some interesting location nearby, t_{min} becomes the application specified latency and the algorithm reduces to continuous location tracking.

Sensor	**Avg. Energy (mJ)**	**Std. Dev.** (mJ)
Fingerprint scan, WiFi+Cell	565	NA
3G Internet	10592	1393
WiFi Internet	738	15
Fingerprint location Cell (FC)	11157	1393
Fingerprint location WiFi (FW)	1303	15

Table 5: Energy used by location primitives averaged across multiple devices. Fingerprint energy is measured only on Asus E600.

6.1 Location Tradeoffs

Table 5 shows energy measurements for the additional sensor primitives required for location. The actual energy consumption depends on which of these primitives are used and how often. We found in our test data that most users move less than 16% of the time during a day. Assuming that motion is detected every two minutes, Figure 1 shows the latency and energy tradeoff for tracking location for six different options. Senergy uses this analysis to select the most appropriate algorithm at runtime. The actual energy use for a user over a day will vary since Senergy will dynamically switch between GPS and fingerprint based location as appropriate. Location accuracy is not considered here since the choice of fingerprint and GPS depends on availability and user permissions.

7. Runtime Algorithm Selection

The quantitative trade-off graphs show that certain algorithms do not yield a measurable advantage in any dimension. For instance, D-CA is very similar to D-ACC in accuracy (Figure 5) but D-ACC provides much lower latency. Hence, we eliminate D-CA, and similarly, W-GPS.

However, many algorithms are better than others on some dimension and no single obvious choice serves all scenarios. For instance, Figure 5a shows that algorithms that are worse on battery use are in fact better in accuracy. For the same energy drain, a lower accuracy algorithm can provide better latency, and vice versa. Senergy selects the appropriate algorithm at runtime when an application makes an API call specifying its requirements.

Once the requirements are available, the selection of the best approach is simply a matter of limiting the search space to those requirements, in the order of priority listed in the API call. As an example, suppose the API

call is `ChangeAlert (Activity.DRIVING, Battery, 5, Priority.LATENCY, 300, Priority.ACCURACY)`. Figure 5b shows that with these battery and latency constraints (shown as gray lines) algorithm D-ACC, D-GPS, and D-AG are the available choices. Considering the last argument requesting an accuracy priority, Senergy selects D-AG, that has the highest accuracy among feasible options. Selecting the algorithm at runtime has the advantages that (1) as context algorithms evolve and tradeoffs improve, such as by using low power processors [17, 26], better algorithms will get selected, transparently benefiting applications, and (2) runtime conditions can be exploited, such as ignoring the battery constraint if the phone is plugged in.

Senergy defaults to maximizing energy efficiency when the API call is under-constrained. For instance, if the third priority request for accuracy was absent from the above API call, Senergy would choose D-ACC, the most battery efficient option satisfying the first two constraints. If over-constrained, Senergy ignores the lowest priority constraints. The API call returns a list of the constraints expected to be ignored based on the algorithm tradeoffs at runtime. In future work, we plan to return confidence intervals with every context change callback, to the extent such error can be estimated. In future work, we plan to return confidence intervals with every context change callback, to the extent such error can be estimated. For example, error estimates could be based on posterior probabilities returned by the inference algorithms or error data from sensor drivers (e.g., Horizontal dilution of precision (HDoP) from GPS) or the sensing service (e.g., fingerprint based location services return a server computed error). Error conditions could also be estimated from sensor operating conditions such as GPS satellite count, user permissions for microphone and camera, etc. that affect the validity of sensor data. Compile time feasibility checks are undesirable because future improvements to the context stack may expand the feasible search space.

7.1 Multiple Simultaneous Applications

If multiple applications are simultaneously active, Senergy considers the constraints jointly and enforces the tightest constraints. Of course, constraint spaces from different applications may not overlap. We do not assume that any one application has a higher priority. Instead, we use a default priority order on the constraints. We first drop any battery related constraints and attempt to satisfy the accuracy and latency requirements alone. If even these are conflicting, we prioritize latency and drop the accuracy requirement. Preferring accuracy and latency over energy is based on the intuition that application functionality may be more important than the recharge interval. Among accuracy and latency, we prioritize latency for timeliness. Other priority orders among the three dimensions may be implemented, such as when applications belong to different priority classes.

8. Evaluation

This section shows how Senergy improves efficiency compared to other API design choices while meeting application requirements. In particular, the proposed API yields multiple orders of magnitude savings compared to the existing context API (`addProximityAlert`).

8.1 Datasets used in Evaluation

We collected over 4,200 usage hours of mobile sensor data from 49 participants, as follows.

Driving Data. Ten people labeled when they were *Driving* for up to 5 days each. Driving trips were primarily commutes with a few side trips. Users tapped a button in our logging application when they entered and exited their car. Intermittent stops due to traffic control and congestion are considered driving. A background service continuously collected accelerometer, GPS, and network fingerprint scans.

Multi-Activity Data. We collected ground truth data for 10 participants for driving, walking, and sitting in one hour guided sessions by accompanying participants and manually recording ground truth on a separate time synchronized device. Each participant carried three phones: in their pocket, backpack, and hand/cup-holder. We use this data to evaluate all of our activity sensing algorithms and application scenarios involving multiple activities.

Routine Location Data. We collected location and accelerometer data from 18 participants running our logging application on their own mobile device for 1 to 12 days. This data set contains a total of 124 days of user traces. To ensure the mobile device batteries last at least a day, we collect data for 5 seconds in every minute. We test location algorithms and applications with this data.

Workday Data. We logged continuous accelerometer data from 11 participants for 6-8 hours on one workday each; a total of 69 hours of participant data. We use this data to evaluate activity applications at extremely low latency settings, which is not possible on the other, larger but duty cycled, datasets.

8.2 API Configurations

We compare Senergy with the other API choices (from Section 3) implemented as follows.

Raw: We implement simple algorithms over raw sensor data, such as checking location periodically to infer if the user is near a desired location. We optimize the frequency of checks based on expected context requirements (specified below for each activity). In theory, developers could implement the best algorithms in Senergy, in which case the energy consumption is the same, unless the user executes multiple background applications where Senergy has the additional advantage of sharing context.

Default: We emulate existing implementations in the Android OS. For instance, for detecting proximity to a set of locations, we use the Android approach of periodically checking location. For activity contexts (e.g., driving, walking)

that have no current Android API, we created a default algorithm that senses continuously with the lowest latency analogous to the implementation of `addProximityAlert` .

Fixed Modes: In fixed mode, the system offers multiple defaults. We implement three representative modes that each prioritize one of energy, accuracy and latency.

Fixed-E: The Fixed-E mode prioritizes energy efficiency. It assumes a two minute latency is acceptable for all applications. It uses the lowest energy inference algorithm that is sufficiently accurate to be included in the OS. For instance, location tracking uses the low power sensors every two minutes to detect movement. If the user is moving, it updates the location at two minute intervals.

Fixed-A: The Fixed-A mode prioritizes accuracy by using the most accurate inference algorithm available. It assumes that a one minute latency is acceptable for all applications. For instance when tracking location, Fixed-A does not use the low power sensors to first detect user movement, since the errors in movement detection may miss periods of movement and increase the overall error in location.

Fixed-L: The Fixed-L mode prioritizes latency by supplying the lowest possible latency, which on our platforms is 5 s and is the lowest latency at which a-GPS may obtain a location fix. Fixed-L tracks location using the low power sensors to detect movement at the 5 s interval before sensing location.

Senergy configurations: For the purposes of this evaluation, we choose two Senergy configurations for each application out of the wide range of priorities and constraints that developers may specify:

Senergy-S: The application developer expresses one primary priority, either energy, accuracy, or latency, and optionally a quantitative constraint.

Senergy-M: The application developer expresses multiple priorities and constraint values.

We now consider six different applications. Three applications use location context and three use activity context.

8.3 Location Context Case Studies

ClubPoint. After purchasing a mattress, Alice realized she could have saved 15% if she had remembered to use her AAA Club card. Alice writes the ClubPoint application to remind her to show her card at participating stores. She sets latency to 5 minutes (300 s) since she only wants notifications when she stays in a store long enough to buy something. The Senergy-S (single priority) call is:

```
Location[] locations = GetAAALocations();
ChangeAlert(locations,Priority.LATENCY,300)
```

Using the algorithm from Section 6, Senergy-S checks for movement using the accelerometer and if the user is moving, it updates the location at the specified latency.

Alice explores the API further and decides to add a battery budget of 5% daily consumption and request high accuracy as a third priority. This multiple constraint Senergy API call (Senergy-M) for ClubPoint is:

```
ChangeAlert (locations,Priority.BATTERY,5,
Priority.LATENCY,300,Priority.ACCURACY)
```

Because there is a 5% battery budget, Senergy can sense movement more frequently than the default 2 minutes. Since our Routine Location Data indicates that most people move up to 16% of the time over a 24 hour window, Senergy reserves sufficient battery out of the 5% to update location every 300 s for 16% of 24 hours, and uses the remaining budget to check user movement via the accelerometer. Using the energy measurements for location sensing, with a mix of indoor and outdoor locations, as well as low power movement sensing (Tables 4 and 5), we obtain a movement sensing frequency of 54 s. The third requirement of high Accuracy priority is ignored since increasing accuracy by avoiding the error from movement sensing is not feasible within the 5% battery budget.

Figure 9 compares ClubPoint configurations for Raw, Default, Fixed, Senergy-S, and Senergy-M. It plots average energy for 18 participants with their routine movement patterns (Routine Location Data). Log scale is used for clarity. The error bars show the standard deviation in energy usage across multiple participants. Fixed-A, Raw, and Default do not depend on user behavior and have zero standard deviation.

Default, the existing implementation, requires 177% of the battery in a 24 hour day, exhausting the battery in less than 24 hours. All other choices reduce battery drain. Fixed-L (latency mode) reduces energy draw to 73.4%. This savings come from checking for movement every 5 s (the smallest latency) and activating the GPS only when the user is mobile, rather than continuously. Fixed-A (accuracy mode) saves energy because it only checks for location every minute rather than continuously. Fixed-E (energy mode) operates within 4% of the battery, but it is still wasteful since it updates location every two minutes and does not exploit the application's latency tolerance of 5 minutes. Raw checks for location every 5 minutes, but does not use movement sensing first, and thus burns slightly more battery than Fixed-E. Senergy-S provides the lowest energy draw, using only 2.7% of the battery capacity over a 24 hr period. Senergy-M uses more battery, 4.5% because the developer specified up to 5%. Senergy-M exploits the allowed battery budget to offer higher accuracy by checking for movement more often.

SimplySave. Jason installs SimplySave, a coupon application that alerts him whenever he passes by a business that offers a discount. Unlike ClubPoint, SimplySave wishes to detect proximity to participating locations with a lower latency of 60 s, since the user may not spend much time at each location. The Senergy-S API call is

```
ChangeAlert (locations,Priority.LATENCY,60)
```

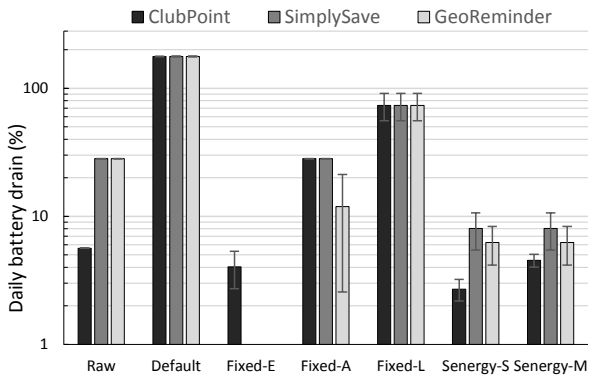


Figure 9: Location context application energy on a log scale with Routine Location Data, averaged over four platforms. Values greater more than 100% indicate the battery is exhausted in less than a day. Error bars show standard deviation in behavior across 18 participants.

Senergy-S updates location every 60 s and optimizes energy over accuracy by first checking for movement every minute and only checking location if the user is mobile. The Senergy-M API call constrains battery budget to 5%:

```
ChangeAlert (locations,Priority.LATENCY,60,
Priority.BATTERY,5)
```

Senergy-M uses 60 s as the latency limit and the optimizes energy. Given latency is the higher priority, the expected energy usage is not below 5% and therefore the battery constraint and any subsequent ones are ignored.

The SimplySave bars in Figure 9 show Fixed-E is not applicable because its two minute latency does not satisfy the application's latency requirements. Default, Fixed-A, and Fixed-L for SimplySave do not use any of the application specific information and thus are the same as ClubPoint. The Raw implementation checks location every minute, resulting in 28% battery drain. Senergy-S SimplySave however uses more energy, 8.04%, compared to ClubPoint, since it delivers the lower 60 s latency. Senergy-M and Senergy-S behave the same for SimplySave because the additional constraints from the developer are not achievable.

GeoReminder. Jim uses GeoReminder to add a location based reminder to pick up a book when he passes the bookstore on his way to the bus stop. GeoReminder requires low latency because Jim will only be near the store briefly. The key difference from the previous scenario is that Senergy only needs to detect one location. Even though GeoReminder needs low latency, Senergy only needs to update location at that latency when the user is close to the desired location. For instance, if Jim lives 10 miles away from the store, then for a large part of the day, Senergy only updates location every 10 minutes, assuming it takes at least 10 minutes to move 10 miles. The API calls are the same as for SimplySave. Figure 9 shows Senergy uses lower energy on average for GeoReminder than for SimplySave.

8.4 Activity Context Case Studies

This section presents three activity context applications. For the Raw API, the developer must implement an activity inference algorithm, which could be the same as the one in Senergy. Hence we omit comparing to the Raw option.

DriverMode. DriverMode activates a driver-mode user experience on the phone when it detects the user is in a moving vehicle. For example, it suppress non-critical notifications, turns on voice only, and allows family members and close friends to observe that the user is driving. A simple Senergy-S API call is

```
ChangeAlert (Activity.DRIVING,
Priority.ACCURACY)
```

that only asks for high accuracy. Senergy-S uses its default latency (2 minutes) and uses the highest accuracy algorithm, D-AG within that latency. Another developer may decide that the application should act quickly once the user starts driving, and use a more sophisticated call (Senergy-M) specifying three constraints, prioritized: latency 60 s, 5% of the battery per day, and high accuracy as

```
ChangeAlert(Activity.DRIVING,Priority.LATENCY,
60,Priority.BATTERY,5,Priority.ACCURACY)
```

Senergy-M checks for driving activity every minute but will use the lower energy algorithm D-ACC, since D-AG is not feasible within the requested battery budget. The third priority requirement, accuracy, is ignored. The Default approach continually senses and infers activity, similar to what existing OSs do for location context. Fixed-A checks for activity once every 60 s, Fixed-E checks every 120 s, and Fixed-L every 5 s.

Figure 10 compares the energy use for all API choices for DriverMode. While Default and Fixed-L use more than 100% battery and are impractical for real deployment, Fixed-A and Senergy-S both provide high accuracy at significantly lower energy, due to increased latency. Specifying a battery constraint of 5% with Senergy-M reduces battery consumption to 3.6% by switching the algorithm to D-ACC from D-AG, which degrades accuracy (see Figure 5). Fixed-E goes even lower due to increased latency.

RadioGuide. RadioGuide publishes free local radio station schedules. In return, users allow the application to anonymously track when they drive and listen to the radio, to aid radio stations in optimizing their schedules. Because radio stations are most interested in drives longer than 5 minutes, RadioGuide sets its latency to 5 minutes. The Senergy-S API call is:

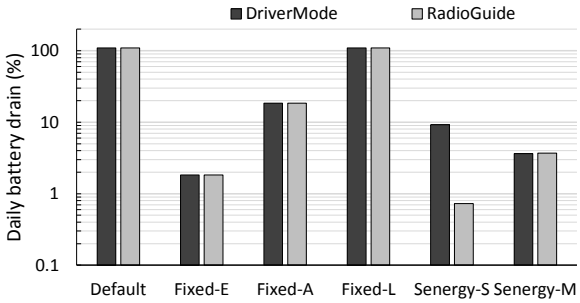


Figure 10: DriverMode and RadioGuide energy with Driving Data.

```
ChangeAlert (Activity.DRIVING,Priority.LATENCY,
300)
```

Senergy-S checks for driving every 5 minutes. Since energy and accuracy are unspecified, Senergy-S defaults to saving energy and uses the lowest energy algorithm, D-ACC. The Senergy-M configuration uses the same latency, restricts battery use to 5%, and requests high accuracy:

```
ChangeAlert(Activity.DRIVING,Priority.LATENCY,
300,Priority.BATTERY,5,Priority.ACCURACY)
```

Senergy determines that both D-ACC and D-AG are feasible within the allowed battery budget at this latency, and chooses D-AG since RadioGuide specifies high accuracy.

Figure 10 compares the energy use for the various API choices. Senergy-M does use less energy than the other APIs, but more than Senergy-S, since it improves accuracy within the developer specified budget.

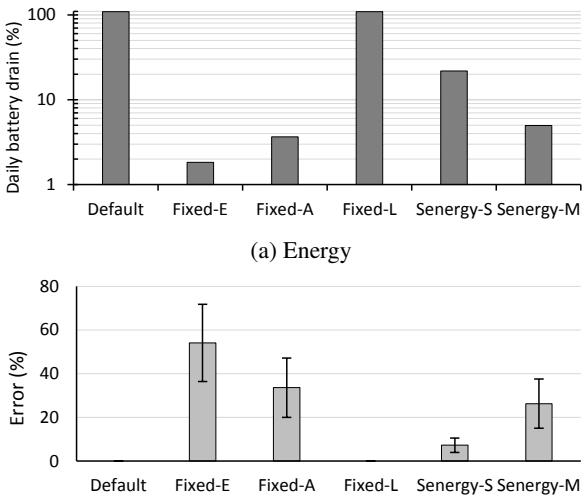
FitnessTracker. FitnessTracker counts a user's daily steps to estimate calorie use. It is representative of mobile applications that track fitness related activities to help motivate a more active lifestyle [6]. FitnessTracker wants a callback whenever the OS detects walking and will then access the accelerometer data to start counting steps until the user stops walking. To not miss short walks, the developer requests a 10 s latency with the Senergy-S API call:

```
ChangeAlert(Activity.WALKING,Priority.LATENCY,
10)
```

Senergy senses every 10 s using the W-ACC algorithm which has the lowest energy. To control battery use, the FitnessTracker Senergy-M call includes a 5% battery limit as the first priority constraint, keeps latency as its second priority, and requests high accuracy:

```
ChangeAlert(Activity.WALKING,
Priority.BATTERY,5,Priority.LATENCY,10,
Priority.ACCURACY)
```

Senergy computes that for this battery constraint, the fastest



(a) Energy

(b) Error

Figure 11: FitnessTracker energy and accuracy trade-offs using Workday Location Data.

it can sense is every 44 s using the lowest energy walking detection algorithm, and thus ignores the lower priority constraints of latency and accuracy. The other modes use their fixed settings.

Figure 11a compares the energy use. While in RadioGuide, the latency goal was to simply satisfy the applications latency requirement, in FitnessTracker, the actual latency affects the accuracy of walk time and step counting. Figure 11b plots accuracy of walk detection. Lower latency does improve the accuracy significantly, especially for two users in our Workday Location Data who take many short walks (not shown), though at the expense of extra energy. Default and Fixed-L do not miss any short walks (no error). The error inherent to the walk detection is not included since it is same across all API choices.

8.5 Multiple Simultaneous Applications

Finally we illustrate the savings when multiple context tracking applications are running simultaneously. As a baseline, we assume that each application was written by an expert and was individually optimized using the same techniques as Senergy. However, using Senergy is still advantageous. With Senergy, energy is spent on the sensors once and the sensor data is used to compute all context outputs needed by the various applications as opposed to each application accessing the sensors at the time that it wakes up. Secondly, if the algorithm used for one application suffices for others, even though it uses a different sensor, then Senergy does not use the other sensors or algorithms (e.g., if one application is repeatedly using GPS while another one was using the accelerometer to detect movement before activating GPS, then Senergy turns off the accelerometer since GPS is anyway being used). Finally, if one algorithm senses with more ac-

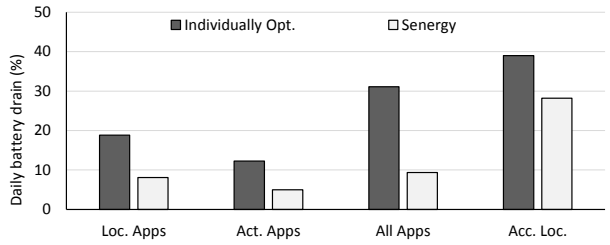


Figure 12: Energy savings when individually optimizing each application and using Senergy. Each case meets the same latency and accuracy requirements. The geometric mean saving is 50.61% compared to individual optimization.

curacy or lower latency for any one of the applications, all other applications benefit from it.

Figure 12 shows the savings for four illustrative combinations of applications. *Loc. apps* consists of all three location applications (ClubPoint, SimplySave, and GeoReminder described in Section 8.3) executing at the same time. *Act. apps* consists of the three activity context based applications (DriverMode, RadioGuide, and FitnessTracker) executing at the same time. *All apps* denotes all six of these applications. Senergy-M satisfies all these requirements in all instances. For example with the location applications, Senergy-M satisfies all of the latency and accuracy constraints in all three groups by using the accelerometer to first detect movement, and therefore delivers substantial energy gains of about 70% compared to each of the three applications waking up at its specified latency and reading the accelerometer.

The fourth set of applications, *Acc. loc.* consists of the same three location applications where ClubPoint and GeoReminder use the same settings, but a modified version of SimplySave that instead requests the highest accuracy setting. This setting requires using GPS rather than sensing with the accelerometer first. This fourth set demonstrates the savings when Senergy-M turns off the accelerometer sensor, which although best to use for ClubPoint and GeoReminder, is redundant with the GPS which high accuracy SimplySave now is using. Comparing *All Apps* with *Acc. Loc.* shows that even with a relatively higher demand energy application, Senergy optimizes by reusing the GPS reading, and eliminating the accelerometer reading in the other applications with lower demands.

In summary, using these realistic applications and mixes of applications, we show that with an expressive, flexible API the OS approach implemented by Senergy can tradeoff energy, accuracy, and latency for both individual applications and for multiple applications executing concurrently. Senergy offers programmers a simple and intuitive API that does not require understanding the underlying runtime optimizations. We show that given single and multiple application requirements, the system meets the requirements and delivers reduced energy consumption by orders of magnitude compared to the state-of-the-art APIs in current mobile operating systems.

9. Discussion

Based on our experiences designing and implementing this API, we discuss a number of issues that future systems should consider.

Hardware Architectures. New mobile-processors offer an additional smaller, lower-power core, that can run certain context-sensing tasks much more efficiently [17, 26]. New hardware broadens the range of choices available to Senergy and it may consequently satisfy more application constraints. The tradeoff space should be characterized to include these options. The API does not change, but applications transparently benefit from hardware advances. If hardware advances make new types of context and activities feasible, such as identifying human voices, the API again does not change but starts delivering these additional context types to applications that request them.

Predictive and Historic Context. Senergy does not maintain historical state. Any long term modeling must be performed by the application. However, if Senergy records context, it could learn user behavior models and use them to optimize context sensing. If Senergy records context over time and users give applications access to their historical data, it may empower applications to be more useful immediately after install. For example, if the user installs a fitness application, and Senergy has already been tracking activity context for another reason, it can give this historical information to the fitness application, which could immediately compute fitness levels and adjust the user experience based on this history. Furthermore, if users provide ground truth labels the system can use auto-tuning approaches to adapt itself to a specific user's environment and behavior.

Reporting Latency. We defined latency as the delay in *detecting* and *reporting* a context change to the application. Additional optimization opportunities arise by decoupling the detection and reporting latency. For instance, to track location changes at 5 minute latency, an efficient algorithm may obtain network fingerprints every 5 minutes but not contact the Internet based location server to convert the fingerprints to location coordinates until the device is plugged in, resulting in a reporting latency of several hours. This configuration may suffice for some applications, for example, if the application displays a map of the locations the user visited over the past week. Since capturing the fingerprint is low energy, this optimization will yield significant energy savings. The API would evolve to add a *reporting* latency.

10. Related Work

Prior research has recognized the energy efficiency challenge for continuously sensing context. Several groups have created low-power strategies such as combining location requests, adaptive sampling, and chaining low-power sensors with higher power ones [3, 7, 14, 18, 24, 40], as well as ap-

plication specific energy optimization techniques [35]. Prior work has explored energy-efficient approaches to detecting human activities, such as walking, resting, and meeting [38]. In Jigsaw [19], energy spent on inference is minimized by suppressing the higher energy stages in the inference pipeline using lower energy stage results. Our goal is to encapsulate such techniques for energy optimization underneath a developer friendly API, so developers may benefit from this work without being tied to the underlying methods that may evolve over time.

A few researchers have suggested language support for optimizing energy [5, 33]. In the Eon programming language and runtime [33], programmers express quality of service (latency) and energy constraints and the runtime system adjusts based on available energy. With energy types [5], programmers encode the expected energy requirements by adding energy hints (e.g., low vs high power) to a function. The runtime can use these hints to, for example, throttle a CPU to a low power mode when a function with that designation runs. While it may be possible to implement our API in extended versions these languages, our current implement considers a more complicated trade-off space (e.g., energy, accuracy and latency) than these systems address.

Other systems have also proposed techniques to facilitate context based programming. Kobe [4] considered the latency, energy, and accuracy tradeoff for mobile sensing but for designing inference algorithms. Kobe accepts training data from the developer and generates an optimized inference pipeline, using multiple classifier configurations and cloud offload. SymPhoney [11] accepts a data flow graph from a developer that specifies low level operations such as sensing, feature extraction, and classifiers, and then determines an optimal resource allocation. Senergy could use such methods internally for designing its context sensing algorithms. While Kobe and SymPhoney are targeted at developers building or specifying their own machine learning algorithms, Senergy exposes ready to use context, completely decoupling the inference algorithm details from applications. Code in the Air [29] and the Context Toolkit [31] enhance programmer productivity through re-usable and energy efficient context sensing code but are based on the catalog approach without exposing battery and performance trade-offs. SeeMon [12] and Orchestrator [13] optimize resource use by converting context queries from multiple applications into lower layer sensing and processing primitives, and then selecting the most efficient set of such primitives to satisfy all queries. While this work optimizes energy use, it does not expose battery and context quality trade-off to the developer through a flexible API.

Other related APIs include location APIs specified by W3C [25] and existing mobile OSs, and cloud based location tracking APIs for backend processing [10]. These APIs do not consider the energy efficiency challenges that we address.

In summary, our approach allows developers to specify latency, accuracy, and battery requirements for their context needs. The novel aspect of our approach is that it provides the OS and runtime with the information necessary to optimize resource use without requiring application developers to understand energy intricacies or inference algorithms.

11. Conclusions

We identified the LAB abstraction and showed how to use it to implement energy-efficient continuous context sensing and how it improves programmer productivity. We described a prototype implementation using 22 activity and location tracking algorithms. We illustrated how the Senergy runtime uses the energy, latency, and accuracy requirements specified by applications and the algorithm tradeoffs to deliver energy efficient context sensing under a wide variety of accuracy and latency requirements. We showed for six realistic applications, how Senergy uses a small amount of application flexibility to reduce the battery drain to much more practical levels, compared to using existing APIs. The resulting system gives application developers efficient context without becoming experts in energy optimization or context inference. Whereas in most current mobile systems context is too costly in energy to use regularly, we believe that with Senergy's significant reductions in battery use for continuous context tracking that developers will be able to deliver new context-aware applications that delight.

References

[1] N. Banerjee, A. Rahmati, M. D. Corner, S. Rollins, and L. Zhong. Users and batteries: Interactions and adaptive energy management in mobile systems. In *Ubicomp*, 2007.

[2] G. Challen and M. Hempstead. The case for power-agile computing. In *HotOS*, 2011.

[3] Y. Chon, E. Talipov, H. Shin, and H. Cha. Mobility prediction-based smartphone energy optimization for everyday location monitoring. In *Sensys*, 2011. ISBN 978-1-4503-0718-5.

[4] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *ACM SenSys*, 2011.

[5] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. *SIGPLAN Not.*, 47(10):831–850, Oct. 2012. ISSN 0362-1340.

[6] S. Consolvo, D. W. McDonald, T. Toscos, M. Y. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, R. Libby, I. Smith, and J. A. Landay. Activity sensing in the wild: A field trial of ubifit garden. In *CHI*, 2008.

[7] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox. Enloc: Energy-efficient localization for mobile phones. In *IEEE Infocom*, pages 2716 – 2720, April 2009.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[9] E. Ertin, N. Stohs, S. Kumar, A. Raij, M. al'Absi, and S. Shah. AutoSense: Unobtrusively wearable sensor suite for inferring the onset, causality, and consequences of stress in the field. In *SenSys*, 2011.

[10] M. Haridasan, I. Mohomed, D. Terry, C. A. Thekkath, and L. Zhang. Startrack next generation: A scalable infrastructure for track-based applications. In *OSDI*, 2010.

[11] 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. ISBN 978-1-4503-1169-4.

[12] 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. ISBN 978-1-60558-139-2.

[13] 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 *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 135–144, 2010.

[14] D. H. Kim, Y. Kim, D. Estrin, and M. B. Srivastava. Sensloc: Sensing everyday places and paths using less energy. In *ACM SenSys*, 2010.

[15] M. B. Kjaergaard, J. Langdal, T. Godsk, and T. Toftkjaer. Entracked: Energy-efficient robust position tracking for mobile devices. In *MobiSys*, 2009.

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

[17] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. In *ASPLOS*, 2012.

[18] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys*, pages 285–298, 2010.

[19] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The Jigsaw continuous sensing engine for mobile phone applications. In *SenSys*, 2010.

[20] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu. Speakersense : Energy efficient unobtrusive speaker identification on mobile phones. *Pervasive Computing*, 6696:188–205, 2011.

[21] M. Mun, D. Estrin, J. Burke, and M. Hansen. Parsimonious mobility classification using gsm and wifi traces. In *HotEmNets*, 2008.

[22] S. Nath. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Mobisys*, 2012.

[23] T. Nick, E. Coersmeier, J. Geldmacher, and J. Goetze. Classifying means of transportation using mobile sensor data. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 2010.

[24] J. Paek, J. Kim, and R. Govindan. Energy-efficient rate-adaptive gps-based positioning for smartphones. In *MobiSys*, New York, NY, USA, 2010.

[25] A. Popescu. Geolocation api specification. `http://www.w3.org/TR/geolocation-API/`.

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

[27] M. Rabbi, S. Ali, T. Choudhury, and E. Berke. Passive and in-situ assessment of mental and physical well-being using mobile sensors. In *Ubicomp*, 2011.

[28] K. K. Rachuri, M. Musolesi, C. Mascolo, P. J. Rentfrow, C. Longworth, and A. Aucinas. Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In *Ubicomp*, 2010.

[29] 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. ISBN 978-1-4503-1207-3.

[30] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using mobile phones to determine transportation modes. *ACM Trans. Sen. Netw.*, 6(2):13:1–13:27, Mar. 2010. ISSN 1550-4859.

[31] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM. ISBN 0-201-48559-1.

[32] T. Sohn, A. Varshavsky, A. Lamarca, M. Y. Chen, T. Choudhury, I. Smith, S. Consolvo, J. Hightower, W. G. Griswold, and E. D. Lara. Mobility detection using everyday gsm traces. In *Ubicomp*, 2006.

[33] J Sorber, A. Kostandinov, M. Garber, M. Brennan, M.D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems, In *SenSys*, 2007.

[34] S. P. Tarzia, R. P. Dick, P. A. Dinda, and G. Memik. Sonar-based measurement of user presence and attention. In *Ubicomp*, 2009.

[35] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. Vtrack: Accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys*, 2009.

[36] TI. OMAP 5 mobile application platform, 2011.

[37] A. B. Waluyo, W.-S. Yeoh, I. Pek, Y. Yong, and X. Chen. Mobisense: Mobile body sensor network for ambulatory monitoring. *ACM Trans. Embed. Comput. Syst.*, 10(1):13:1–13:30, Aug. 2010. ISSN 1539-9087.

[38] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *MobiSys*, 2009.

[39] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Mobisys*, 2012.

[40] Z. Zhuang, K.-H. Kim, and J. P. Singh. Improving energy efficiency of location sensing on smartphones. In *MobiSys*, 2010.