“Doing nothing well": OS-Application coordination for energy saving

Kalyanaraman Shankari
David E. Culler
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EEUS-2016-119
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-119.html

June 6, 2016

Acknowledgement

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

“Doing nothing well": OS-Application coordination for energy saving

Kalyanaraman Shankari
EECS Department
UC Berkeley
shankari@eecs.berkeley.edu

David Culler
EECS Department
UC Berkeley
culler@cs.berkeley.edu

Randy Katz
EECS Department
UC Berkeley
randy@cs.berkeley.edu

ABSTRACT

Mobile phones do “nothing well" at last. A stock iPhone 6 has essentially no battery drop overnight, while a stock Nexus 6 running Android 6.0 loses only 6% battery in that time. However, this is achieved by severely restricting background operation: iOS has always restricted the set of operations performed in the background, while Android 6.0 doze mode forces apps to perform their background operations in periodically scheduled maintenance windows. In this paper, we explore a technique for app-OS cooperation to manage background processing. We use the new geofencing APIs, which notify the app when the user leaves a defined region, to turn off location data collection when the user is "loitering" and restart automatically when she starts moving. With this change, we are able to drop the power drain for automatic monitoring of location and activity over a day by 35% to 82% on iOS and up to 100% on Android, depending on the accuracy and sampling rate. The tradeoff is that the error at the start of a trip is higher: we get no points until we exit the geofence, and the first few points after exiting may be at a lower sample rate. We present a summary of the challenges encountered in the practical implementation of our open source library for geofenced duty cycling, particularly on iOS, and the techniques that we used to overcome them. This evaluation, discussion and library can provide a practical design guide for researchers who plan on writing their own sensing applications.

1. INTRODUCTION

Cell phones are ubiquitous, both in the developing and developed world. There are claims that more people have cell phones than toilets. Increasingly, these will be smart phones.

Ever since the early smart phones were introduced, researchers have been been interested in the opportunities for gathering data by using smartphones as sensor platforms. But cell phones are not just a collection of sensor in a convenient package. They are also devices that provide real utility to users - that’s why the adoption rates are so high. This means that users are sensitive to high rates of power drain, so energy efficient sensing has been a research focus for almost as long.

Historically, the two main mobile phone platforms - iOS and Android - have approached the power/utility tradeoff differently. iOS has prioritized user interaction and eschewed background operation, and indeed, multi-tasking, while Android has provided a more traditional, multi-threaded, pre-emptive operating system. However, as the platforms mature, they have realized (1) background operation is critical to providing a good user experience; (2) background operation done poorly is an energy hog; and (3) application developers are rushed and will rarely take the time to optimize their background operations. Both platforms have converged towards a model in which background operations are permitted, but with OS-imposed restrictions that aim to manage the associated power drain.

At the same time, much of the prior research work in this space seems to have made its way into the shipping platforms. For example, the recommended Location APIs on Android no longer require you to choose a provider. The OS will dynamically choose it for you based on the specified accuracy. Similarly, iOS automatically and continuously tracks user activity, whether requested or not, and manages the power drain by using a separate low power co-processor to offload the collection and processing of sensor data from low power sensors.

All of these changes raise some practical concerns for the developer or researcher who intends to build a continuous sensing app today.

Is continuous sensing on mobile phones a solved problem?

In other words, what are the current OS optimizations, how well do they work, and can they be improved?

This paper is our attempt to answer the questions above in the context of tracking mobility patterns - an application which requires the use of the expensive location sensor.

1.1 Contributions

1. We explore the current continuous sensing APIs on two different mobile platforms - Android and iOS

- and document their restriction on background operation.
2. We design a sensing regime that works with the restrictions on background operation on both platforms, highlight the design challenges encountered, and evaluate the selected regime at various accuracy/power points.
3. We build a simple, 3-state model for the power drain at the accuracy points evaluated, and use data from the American Time Use Survey to generalize the results to a wide variety of usage patterns.

The paper outline is as follows: sections 2 provides an overview of the platform APIs, section 3, compares our solution to the related work, 4 outlines the motivation for our approach, section 5 is a brief evaluation of the power/accuracy tradeoffs over a small set of mobility patterns, section 6 covers the construction of a simple 3-state model and its application to a wide variety of mobility patterns, and section 7 concludes with a discussion of future work.

2. BACKGROUND

In this section, we briefly summarize the current support for both background scheduling and location detection on both the Android and iOS platforms, as they relate to continuous sensing.

2.1 Android

2.1.1 Background scheduler

The Android OS provides a fairly standard pre-emptive scheduler. In particular, processes *can* be scheduled to run at time based intervals (every 30 secs) in the background. The OS provides specialized frameworks for co-operative scheduling - the `SyncAdapter/JobScheduler` interface for batching network operations is an example. The recent 6.0 Marshmallow release has moved from suggesting to forcing cooperative scheduling using *doze mode*. This is a low power mode that is activated when the phone is not plugged in, and has been inactive for a while (screen off, stationary). While in this mode, the OS suspends the regular scheduler, including processes scheduled by time, and performs all activities in regular maintenance windows.

2.1.2 Location APIs

The location interface consists of a "classic" `LocationManager` API which provides a time based access to a variety of location sensors such as GPS and the network, and what appears to be a context sensitive, rate adaptive `fused` API supplied through Google Play Services (GMS). For either API, a time filter can be specified to regulate the sample rate. The time filter is a hint - updates can be received more or less frequently based on interaction with other apps and the scheduler. GMS also supports a `geofence` API that monitors dwelling within a particular location in very low power mode.

2.1.3 Activity APIs

GMS supports a native, accelerometer-based[16] API for activity recognition that can be the basis for duty cycling based on activity. Applications can register for periodic activity updates, but there are no guarantees that the updates will be delivered at the requested rate.

2.2 iOS

2.2.1 Background scheduler

The iOS background scheduler has the philosophy that power should be conserved for interaction with the user. This results in very impressive battery life on a stock phone, but places severe restrictions on background operation that require creative workarounds.

In general, processes **cannot run in the background**. This means that in general, application *cannot schedule tasks at specified time intervals* unless the app is in the foreground and the user is actively interacting with it. A small set of background operations can be enabled if permitted by the user, including two modes that can be used to request periodic wake ups. `Background fetch` is scheduled locally, but prior testing on iOS7 and iPhone 4 indicated that it is not reliable since the OS would duty cycle it based on network signal strength and user interaction patterns. `Remote push` wakes up the app to handle messages pushed from a server through a messaging service. While it is not guaranteed to be reliable either, it is fairly reliable in practice.

As an aside, the restrictions are severe enough that there is speculation that developers have resorted to playing blank sounds (playing music is a supported background operation) to keep their apps active in the background [14].

2.2.2 Location APIs

Fortunately, location tracking is one of the supported background operation modes. This means that the application can receive location updates even when it is running in the background.

The `standard` Location APIs on iOS do not allow users to specify a provider - instead, similar to the GMS `fused` API, users specify an desired accuracy (best, 10m, 100m, 1km, 3km), and the OS automatically picks a provider or set of providers. The sampling rate is controlled by a distance filter - there is no time filter, maybe because too many developers were using periodic location updates as a background timer.

If an application has requested location updates using the standard API, they will not be delivered if the application has been terminated due to memory pressure. A second `significant location changes` API can restart the app to deliver updates, but it is very coarse and does not support any configuration parameters. Updates are received when the OS determines that there has been a *significant change* - a term with no precise definition. This means that the error model for iOS tracking could include large gaps in tracking for which there is no workaround. Fortunately, this appears to be rare in practice.

iOS also supports a `geofencing` API similar to the one on Android.

2.2.3 Activity APIs

iOS also supports a native activity recognition API that can provide periodic updates. However, the updates are **NOT** delivered when the application is suspended. This means that this API cannot be the basis for duty cycling based on activity, since we cannot reliably detect when the user is in motion again and turn on location tracking.

3. RELATED WORK

This work is at the intersection of several themes in the research literature.

3.1 Academic Literature: Location

The most relevant theme deals with lowering the power of location tracking.

Paek [8] and Entracked [5] assume that the tracking will be continuous, and provide strategies to turn off the GPS intermittently for short periods of time during a trip. Paek [8] uses the requested accuracy and Entracked [5] uses the user's activity. In order to cooperate with the restrictions on background operation in the OS, we explore the ability to stop tracking for large periods of time, perhaps for the majority of the day. In this context, we are closer to the manually launched tracking solutions such as CycleTracks [3] or Biketastic [10], except that in our case, the launching is automated.

The functionality from Bareth [1], which determines location using sources other than GPS, appears to have been incorporated into mobile OSes, and provides the basis for the fused API. In fact, the data collected from Android using the `batterystat` API indicate that even while using high accuracy tracking, the GPS is rarely turned on. But as we can see from Figure 1, there is still a tradeoff between accuracy and power drain, and the power drain of the medium accuracy mode on iOS is still fairly high.

The TAMER project [6] appears to be the academic precursor to doze mode - it automatically interposes itself between the applications and the OS in order to reduce the frequency of background tasks. Our work is complementary to theirs because they want to tame the behavior of badly behaving apps, while we want to make the apps behave well in the first place.

3.2 Academic Literature: Context sensitivity

Chu [2] and ACE [7] explore the use of lower power sensors and smart inference to return the requested data using lower power sensors. So this is similar to Bareth and the existing low power APIs on the phones. We duty cycle on top of that to reduce the power drain of even the lower power sensors.

3.3 Academic Literature: Activity Detection

The activity detection literature has papers ([15] and [13]) on duty cycling for energy efficient sensing, but for ongoing activity recognition instead of location detection. They, particularly Srinivasan [13], also point out that a significant proportion of the power drain in continuous sensing is not the power drain of the sensor, but the power consumed by waking up the CPU to deal with the sensor. These insights provided the motivation for us to explore the technique in the paper.

3.4 Industry

None of these provide any evaluation or implementation details, so they are listed here for completeness.

Google location history [4] is turned on by default on all Android phones. An examination of the data collected in user accounts seems to indicate that it reads the location every minute using medium accuracy. The primary application appears to be place, rather than trip detection, although there are reports that it is combined with activity recognition results to display trips.

Moves [9] is a fitness tracker app for both Android and iOS. In earlier work, we had integrated with Moves for data collection instead of writing our own [12]. Our result was that out of 44 users who installed moves, only 8 retained it for more than 3 months, and they were all Android users. All iOS users uninstalled as soon as the semester was complete. The top three reviews in the app store complain about battery life being impacted. The inability to understand their techniques and to modify them was part of the motivation around designing our own data collection system.

4. DESIGN CHALLENGES

In this section, we describe the motivation and decision challenges involved in building a library for duty cycled data collection on both Android and iOS.

4.1 Motivation

As we can see from Section 5.2.2 and Figure 5, there is significant power drain on iOS for ongoing tracking, even with a large distance filter to throttle updates. The power drain ranges from 1%/hr for the medium accuracy 100m filter to almost 4%/hr for a high accuracy

stat	drain % /hr	reporting service launches/hr
High accuracy, no filter	8.42	2722
Medium accuracy, no filter	0.25	4

Table 1: Comparison of Android sensing regimes when phone is stationary

100m filter (measured separately) even when the phone was stationary on a desk. This would imply a 24 hour power drain of 24% for medium accuracy and 79% for high accuracy even when the location is not changing. The additional power drain for geofencing over stock phone operation is essentially zero.

On Android, the situation is more complex, since the medium accuracy mode uses WiFi, and WiFi scans are suspended in doze mode except for the maintenance windows. Table 4.1 shows that ongoing sensing is effectively duty cycled by default when the phone is in doze mode. So duty cycling would primarily help when high accuracy sensing is needed.

We also considered two other duty cycling approaches from the literature.

1. *Leave the accelerometer turned on and duty cycle other sensors if the user is stationary for a certain period of time.* Unfortunately, this will not work on iOS. Reading the accelerometer and/or activity detection results are not supported background modes. By piggybacking on the location tracking, we can potentially detect when the user has stopped moving, but we cannot detect when the user has started moving again.
2. *Change the location filter properties based on user speed.* Again, this is not likely to help in iOS because changing the distance filter does not appear to appreciably reduce the power drain. Reducing the sampling rate also does not appear to appreciably change the power drain on Android while in medium accuracy, although the power drain is low to begin with. While this might be an acceptable strategy for Android, we wanted to explore a consistent strategy across both platforms in which we just turn everything off.

4.2 Our design and some challenges

In our design, we use a simple two state finite state machine to perform duty cycling.

1. We use location updates to detect when the user is *loitering* or *dwelling* at a location. We will detect loitering even if the user is walking, as long as all movement is within the location radius. So even if the user is walking around the office, she is dwelling in the office.
2. Create a geofence at the current location and turn off all tracking.
3. When the geofence is exited, resume all tracking.

For details of the implementation, we refer readers to the library that we have published on github under a BSD license [11]. Here we will discuss three design challenges and our solutions for each of them.

4.2.1 *Detecting dwelling with a distance filter*

Detecting the end of a trip with a time filter is pretty straightforward. If the user has not moved more than distance d in the past t minutes, then end the trip. This has been the approach taken by most prior work, based on data from GPS devices. But on iOS, the only supported throttling option is a *distance* filter. So once a trip has ended, we will simply stop getting updates. The next update will occur when the user has travelled more than d. Depending on the value of d, this could well be at the start of the next trip, which means that no duty cycling will occur. If there was support for scheduling jobs at a future time, we could schedule a job to be run after t minutes, which would end the trip if there were no recent updates, but as we have seen, that is not a supported background mode. So we use remote pushes, to wake the app up periodically and check to see if the last received location was t minutes ago. Since remote pushes are scheduled on the server, and we do not want to require a network call for each location update, the remote pushes are scheduled to run every hour, with no app triggers or communication.

4.2.2 *Detecting dwelling in the presence of noise*

We originally assumed that the dwell algorithm would be robust to noise because the noise would die down eventually and the geofence would be created. This assumption about the noise model was incorrect, specially in medium accuracy mode. On Android, the medium accuracy collection would sometimes repeat the last known point if it had no new data. This would cause the algorithm to terminate trips whenever we were travelling underground, for example. On iOS, with medium accuracy tracking, spurious, low accuracy points would be generated outside the distance filter, which would cause a continuous set of updates between the current location and the spurious location. Also, we only check for trip end every hour, noise showing up at the wrong time can lead to tracking for an additional hour and wasted power. We address this by filtering both noisy points and duplicates on the the phone before checking for the trip end.

4.2.3 *Geofence creation quirks*

1. iOS will create a geofence but not start monitoring it in the background. So we need to wait for the geofence creation to complete before returning.
2. on iOS, if you are already outside geofence by the time it is created, you won't get an exit. So after

creation, we need to check whether we are inside or outside and transition states accordingly.

3. creating the geofence at the "current" location has some drawbacks - if the current location has low accuracy, we might create the geofence some distance away, and not trigger it while leaving.

4.2.4 *Summary*

To summarize, designing a robust, efficient, cross-platform pervasive sensing app is a significant challenge, full of philosophical differences (background processing), subtle quirks (geofences) and undocumented error models (medium accuracy) that can only be discovered by implementation and testing. We now evaluate our solution and show that it helps with both high and medium accuracy data collection regimes on iOS and with high accuracy data collection on Android.

5. EXPERIMENTAL RESULTS

5.1 Experimental setup

We are exploring techniques to reduce power consumption by duty cycling for long periods of time.

Since the phone OS can dynamically adapt its behavior based on patterns of user activity and interaction, it is not possible to extrapolate from a short sample to behavior over a day. If we want to evaluate the behavior of the phone OS over a day, we need to measure the power drain over the course of a day and ensure that it enters various operating states during that time. Further, since it is not known whether the operating states are deterministic, a direct comparison between data collected on various days is not known to be accurate.

Therefore, our experiment setup consisted of three identical phones for each platform - three iPhone6s and three Nexus 6 phones. We installed the data collection regimes that we wanted to compare on the three phones simultaneously, and carried each set of phones from the same platform in the same pocket.

In order to ensure that the phones had the chance to move through a variety of states, we divided each data collection day into "day" and "night" cycles, each of which was roughly 12 hours long. We also took several short trips throughout the "day" cycle. The trajectories for the trips were not identical, although their cumulative duration was roughly identical, and the "night" cycle occured at different offsets in the day. This means that the data collected across days is not comparable.

5.1.1 *States*

The states that we were trying to exercise as part of the data collection were:

1. **Passive** The phone is not being actively used - it is stationary with the screen off. We expect that the phone will be in this state while the user is sleeping, for example. This corresponds fairly closely with the Android Doze mode. We expected that the phone would be in the passive state for most of the "night" cycle.
2. **Active** The phone is being actively used, but the user is not traveling. This state is key to our evaluation, since we can turn off tracking in this state and reduce background operation. Note that we detect that the phone is active even if the user is walking, as long as she does so within a small radius, like that of a building. We expect that the phone will be in this state for most of the "day" cycle.
3. **Moving** The user is taking a trip while carrying the phone. We expect that the phone will be in this state when we take trips during the day.

5.1.2 *Data collection regimes*

We primarily use the following data collection regimes to explore the range of behavior in each of the states above. Note that the details of the regimes are slightly different on iOS and on Android, since they use different filters. Each of the sampling regimes above is evaluated both with and without geofencing, which gives us six different data collection regimes overall.

1. **High accuracy, fast sampling** (2s on Android, 5m on iOS) (`hafs`)
2. **Medium accuracy, fast sampling** (2s on Android, 5m on iOS) (`mafs`)
3. **Medium accuracy, slow sampling** (30s on Android, 100m on iOS) (`mass`)

5.1.3 *Metrics*

Since our technique trades off power and startup accuracy, we use the following metrics to evaluate the two aspects of the tradeoff.

1. **Power drain** We measure the power drain across different regimes running in parallel on the three phones. We look at both the *final battery level* at the end of 24 hours, and the *power drain in various states* under the regime. The power drain is represented using boxplots - the center line is the median, the box represents the 25th to 75th percentile, the whiskers represent the inter-quartile range (IQR) and outliers are represented by individual points lying outside.
2. **Accuracy** We look at the distance between the actual start of the trip and the geofence exit location. We also inspect the distances between the geofence exit location and the first few points in order to estimate the loss in accuracy at the start of the trip.

Each of the power drain result figures represents data collected over one day to compare regimes against one another. The top graph in each figure represents the

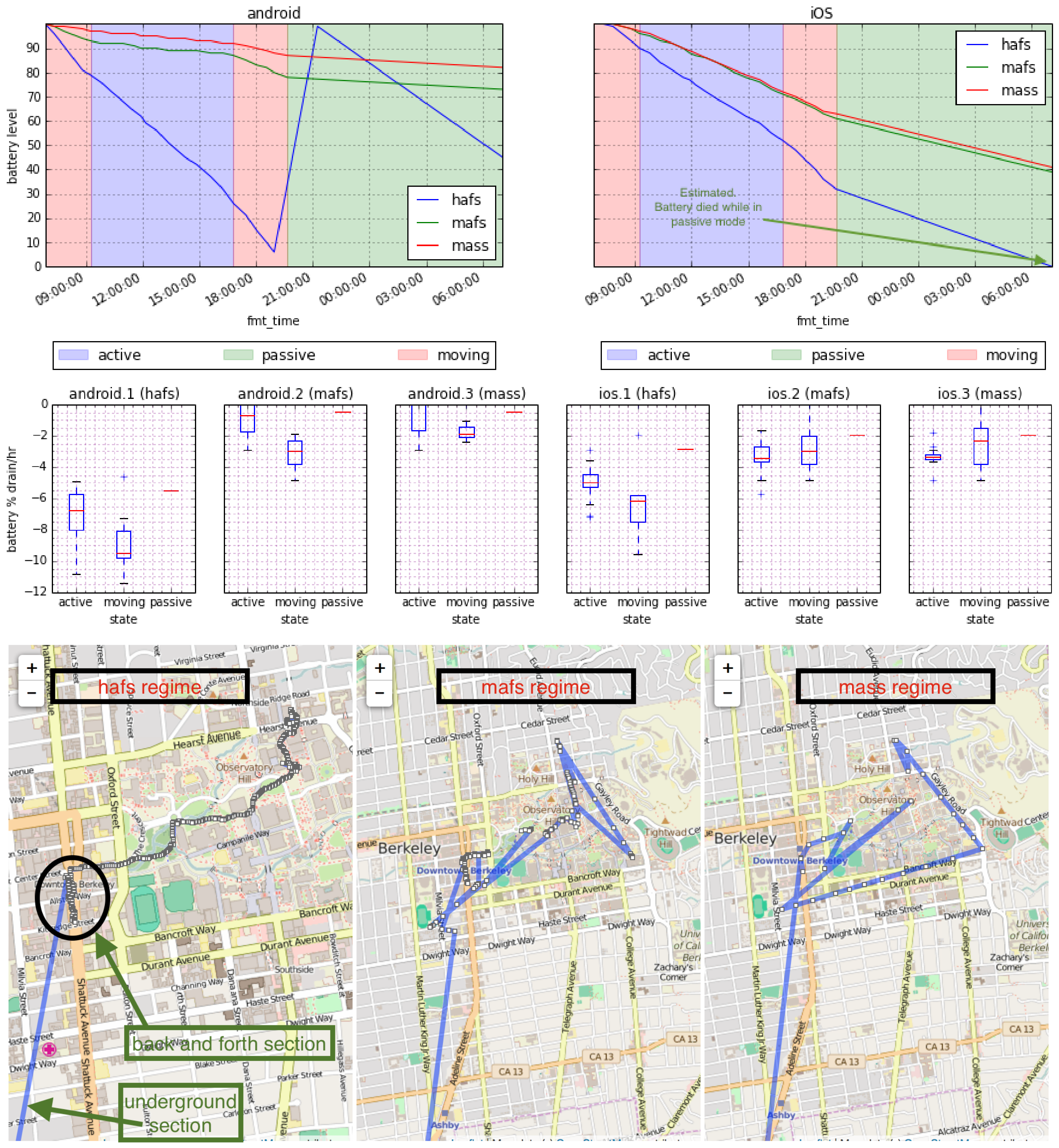

Figure 1: Comparison of three existing data collection regimes with no geofencing. Regimes are high accuracy fast sampling (`hafs`), medium accuracy fast sampling (`mafs`) and medium accuracy slow sampling (`mass`). The top graph shows the change in battery level over 24 hours. The middle graph shows the rate of drain in %/hr in the three states. The bottom map shows the data points collected by each regime and provides an intuition of what the different accuracy levels correspond to.

change in battery level over the course of the day, and the middle graph contains the boxplots for the power drain in the various states. These graphs corrrespond to the power drain metrics above. The third row has a set of maps which provide a visual representation of the accuracy of the data collected.

5.1.4 *Recording measurements*

One of our challenges was to develop a technique for measuring battery levels that would not perturb the measurement. For example, it was not clear that we could use a power meter to measure power drain, since the OS only puts the phone into Doze mode when it is unplugged. Automatically polling for the battery life, in active or passive states, for example, would introduce new background processing at a time when our goal is to reduce or eliminate it. And automatic sampling means that states may overlap with samples, which makes it harder to calculate drain.

Therefore, we used the following low-tech solution to record the power drain.

1. **moving** state: Every 30 minutes, view the battery level on the phone screen and manually record it in a csv file. In addition, record entries at the start and end of every trip so that each time interval has only one state associated with it.
2. **active** state: We use the same technique as the **moving** state. This has the added advantage that it simulates the user interacting with her phone periodically, and ensures that the phone remains in the active state.
3. **passive** view and record the battery level at the beginning and end of the passive period. This ensures that there is no additional interaction with the phone, although it means that it is harder to isolate outlier points.

[p]

5.2 Answers

In this section, we answer the questions that we asked in the introduction.

5.2.1 *Do phones really do nothing well?*

Figures 2, 3 and 4 compare the power drain of continuous data collection against geofenced data collection and no data collection. So the “nd” line on the top-most graph in each figure represents the power drain with no data collection (stock phone) over 24 hours. We can see that the values for iOS and Android are 7% and 11% respectively. This implies a standby time of **12 - 10 days** even if the phone is being carried around and is unlocked every 30 minutes for a few seconds.

5.2.2 *Is continuous sensing on mobile phones a solved problem?*

Figure 1 compares the power drain of three different continuous sampling regimes over the course of the same day. Note that moving from high accuracy to medium accuracy makes a significant difference, with the high accuracy fast data collection running out of battery on both platforms, unlike the medium accuracy sampling. It is also interesting to note that the filter size does not appear to make any difference on iOS - the lines for the fast and slow sampling are almost indistinguishable. On Android, the sampling rate seems to matter primarily during the *moving* state. During the *passive* state, the slopes of the lines are very close, and the divergence in the active state is small.

We can also see this from the boxplots of the power drain rate - in all the Android regimes, the moving rate is noticeably higher than the active rate, which is in turn significantly higher than the passive rate. On iOS, we see a similar pattern for the high accuracy case, but for medium accuracy, there is not much difference in the power drain across sampling rates, or across states within the same medium accuracy regime.

Figure 1 also shows the tradeoff in lower accuracy of the collected data points. The three trajectories shown were recorded at the same time on three identical iphones. As expected, the high accuracy data collection is an accurate representation of ground truth - the duplicated points on Shattuck represents an actual back and forth section of the trip.

While the high accuracy data collection is clearly superior to both medium accuracy data collection regimes, the location accuracy required depends on both the algorithms that are used to process it, and the final application for which it is used. A determination of the optimal accuracy and sampling for different applications is outside the scope of this paper - we content ourselves with evaluating the effect of geofenced duty cycling on each of the three sampling regimes here.

5.2.3 *Does geofencing help?*

Figures 2 shows the effect of geofencing on the power drain with high accuracy sampling. With geofencing, we can obtain high location accuracy during the trip with a power drain that is close to no data collection. In fact, high accuracy data collection is not possible without duty cycling on either platform. The story on both platforms is remarkably consistent - duty cycling makes high accuracy data collection possible.

The picture is less clear if we are willing to tolerate medium accuracy. Figures 3 and 4 illustrate the differing ways in which geofencing affects medium accuracy sensing on the two platforms. Medium accuracy data collection is very efficient on Android - it appears to be similar to the power drain of geofencing. In fact, at the end of the day, the non-geofenced solution actually has a lower power drain than the geofenced solu-

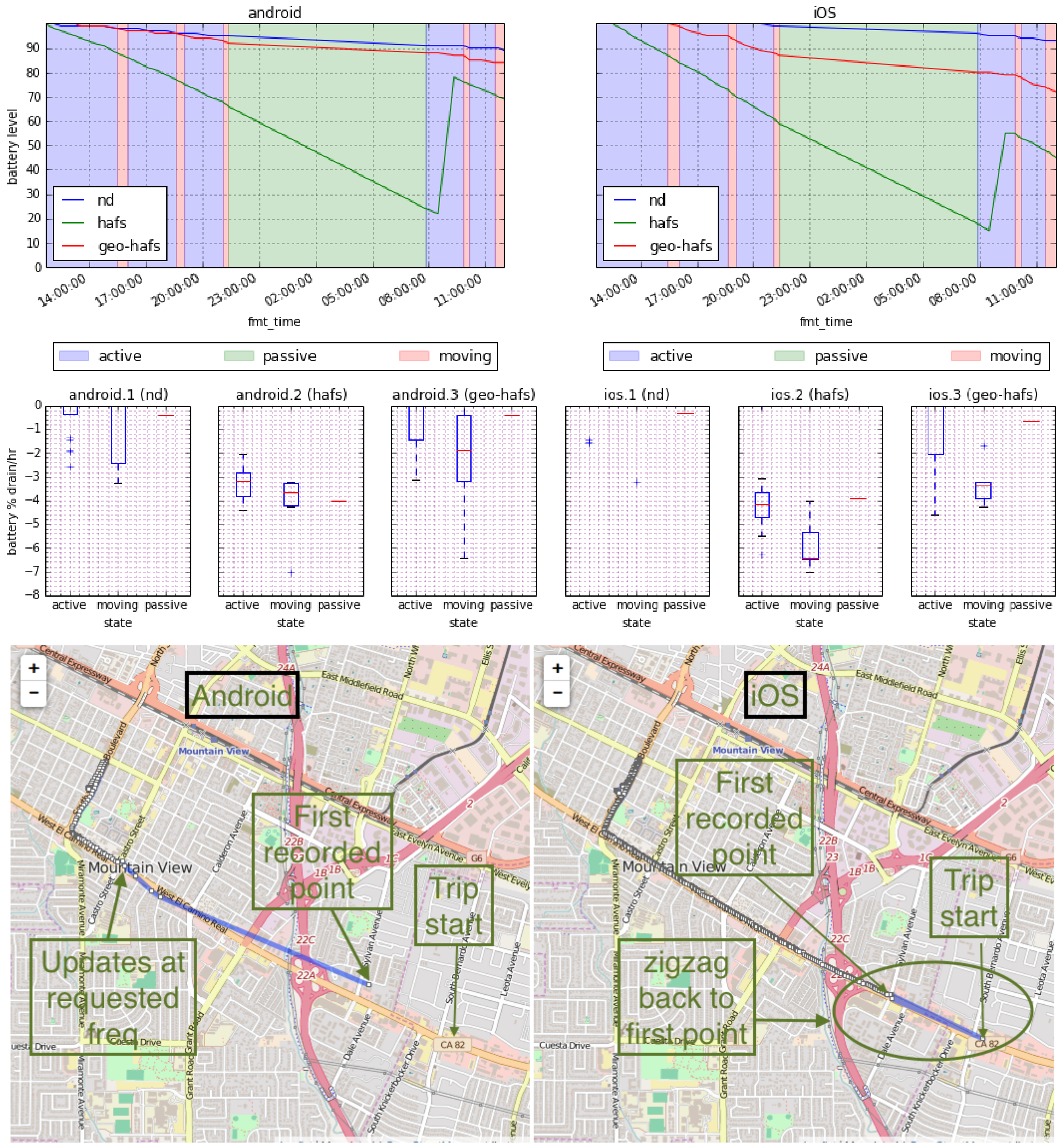

Figure 2: Evaluation of geofencing while collecting data with high accuracy. Regimes are no data collection (`nd`), high accuracy fast sampling (`hafs`) and geofenced high accuracy fast sampling (`geo-hafs`). The top graph shows the change in battery level over 24 hours. The middle graph shows the rate of drain in %/hr in the active, moving and passive states. The bottom map shows the extent of the error on each platform at the beginning of the trip.

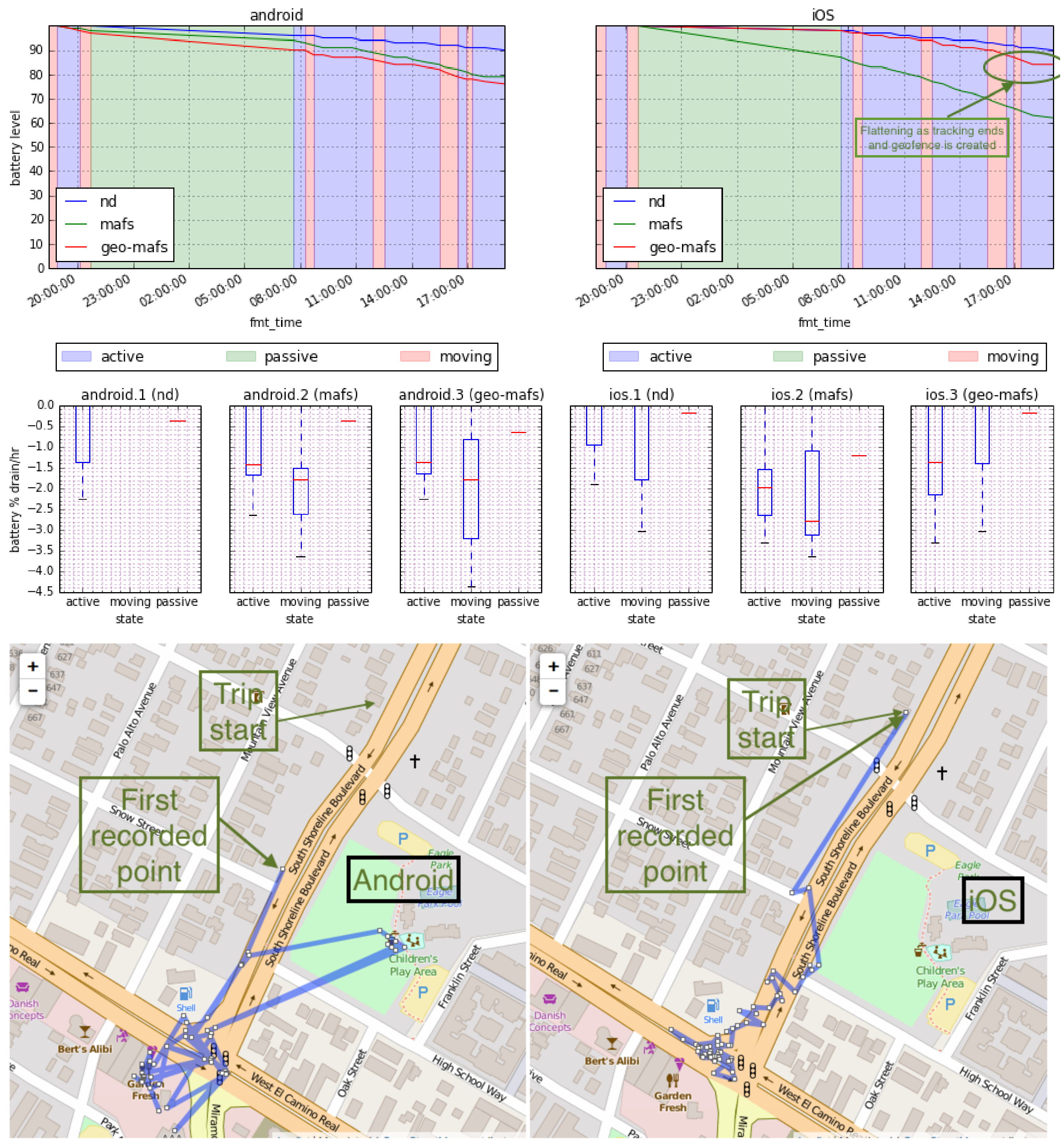

Figure 3: Evaluation of geofencing while collecting data with medium accuracy, but a fast sampling rate. Regimes are no data collection (`nd`), medium accuracy fast sampling (`mafs`) and geofenced medium accuracy fast sampling (`geo-mafs`). The top graph shows the change in battery level over 24 hours. The middle graph shows the rate of drain in %/hr in the active, moving and passive states. The bottom map shows the extent of the error on each platform at the beginning of the trip.

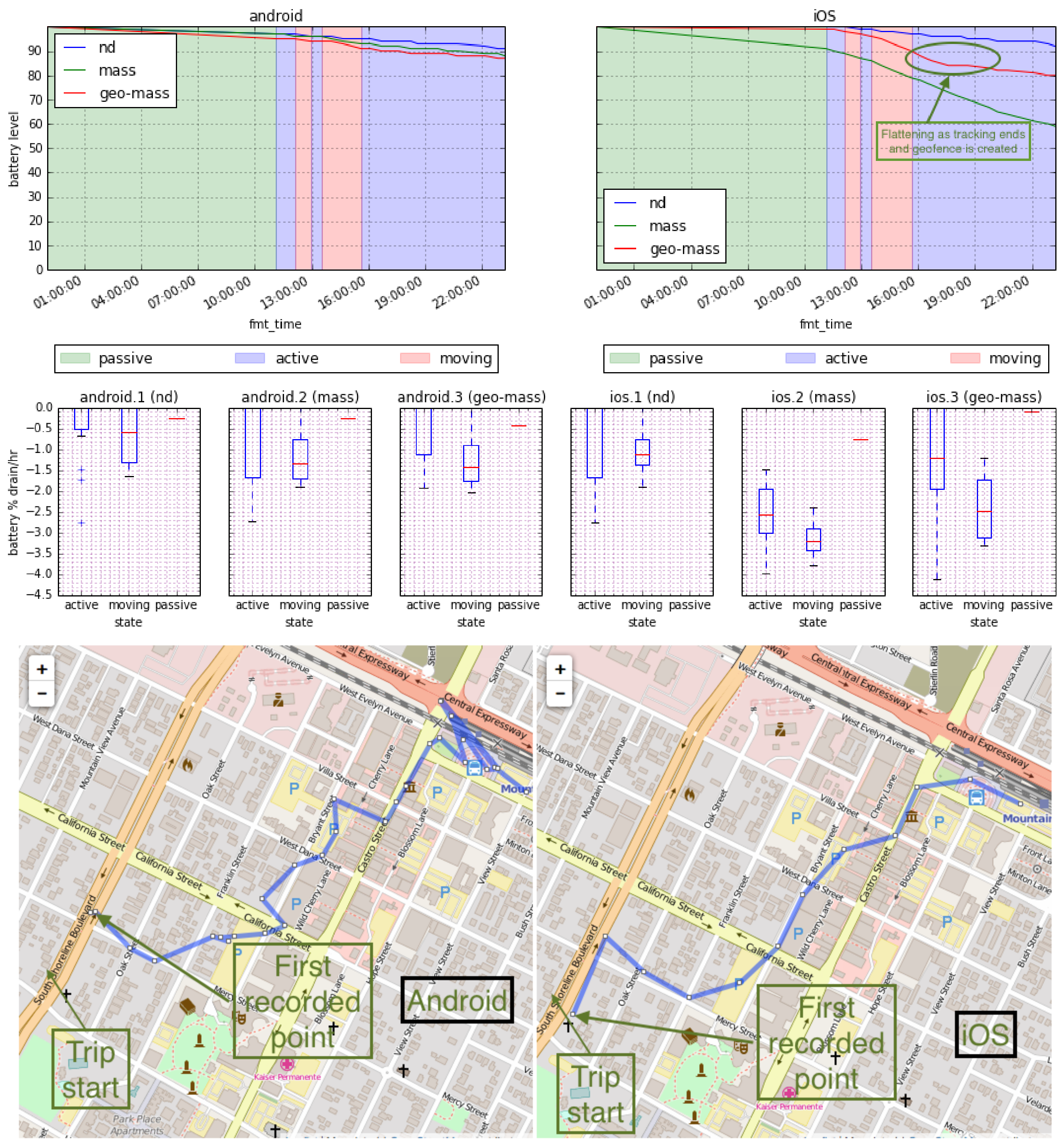

Figure 4: Evaluation of geofencing while collecting data with medium accuracy, and a slow sampling rate. Regimes are no data collection (`nd`), medium accuracy fast sampling (`mass`) and geofenced medium accuracy fast sampling (`geo-mass`). The top graph shows the change in battery level over 24 hours. The middle graph shows the rate of drain in %/hr in the active, moving and passive states. The bottom map shows the extent of the error on each platform at the beginning of the trip.

tion. From the boxplot, we can see that power drain in the active state is almost identical for both geofenced and non-geofenced operation. In the passive state, the power drain is actually slightly higher with the geofence. While the passive state has only one data point, the active state has several points and the result is consistent across both sampling rates. It looks like Android has figured out how to perform continuous, medium accuracy data collection very cheaply.

The story is very different on iOS, where geofencing is significantly cheaper than continous sensing. Even with medium accuracy, the difference in battery level between geofenced and non-geofenced operation at the end of 24 hours is between 25 and 30%. The boxplot shows that geofencing is essentially free in the passive state. The drain appears to be high in the active state, but at least part of that is because we are unable to detect the trip end immediately and have to wait until the next hour to stop tracking. A set of short trips causes us to spend a lot of time in *active but tracking* state, which increases the power drain. For example, note that the big drop in battery level during the moving state in Fig. 4 continues beyond the end of the trip, before flattening out as the tracking stops and the geofence is re-established.

Finally, although geofencing enables high accuracy data collection during the trip, it loses some accuracy at the start of the trip. Location tracking is started only after the geofence boundary has been crossed, so the points traversed to reach the boundary are lost. An interesting observation that we can see what looks like rate adaptive GPS tracking on Android in the `hafs` regime - after the geofence is exited, the points are generated at expontentially shorter distances, until they settle into the configured frequency. We do not see similar behavior in iOS high accuracy mode, and the low accuracy data on both platforms is so noisy that it is hard to determine what the correct points are, let alone their frequency.

6. MODELING

The experimental results above are for a small, restricted set of travel patterns. Since our approach consists of lowering the power drain in the active and passive states, its performance is heavily dependent on the time spent in each state - if a user spends the whole day travelling, there will be no difference in power drain between our solution and the continuous data collection solutions.

So in order to complete our evaluation, we need to extend the results from the four data collection days above to typical days in the life of the general population.

We do this by building a model of the power drain in each state for different regimes and applying it to a large set of user activity patterns collected as part of the American Time Use Survey (ATUS).

The ATUS is a publicly available dataset collected by the Department on Labor that consists of a set of activity diaries which include coded activities, their start and end times and their duration for a randomly selected sample of the population. The 2014 ATUS data contains data from **11592** individuals, whose activities are coded into the 17 major codes. The code include both sleeping (code 1) and all forms of transport (code 18) - there is no category for code 17. For simplicity, we assume that people are interacting with their phones at any time that they are sleeping and not travelling, so we can easily map the major codes to our states.

Next, we combine the data collected above to build a composite model that has a power drain coefficient for each state. We do this by combining the entries from all time periods when that regime was active and calculating the overall mean. In particular, each of the continuous sensing regimes is modelled by considering data from the *accuracy versus sampling rate* data, the *ongoing regime* and the "moving" sections of the *geofenced regime.* Note that this results in 5 coefficients, since the drain for the geofenced modes is a combination of geofencing for active and passive, and a selected sensing mode for moving.

The resulting model is shown in Figure 5. The model appears to be consistent with our observations in Section 5 - the biggest power drain is in the `hafs` regime, geofencing is essentially free on iOS, and geofencing doesn't buy much on Android.

We can then estimate the power drain over the day for every user for a particular regime by: (1) mapping the activity codes to states, (2) summing up the durations in each state to obtain the percentage of the day spent in each state, and (3) multiplying by the coefficients and summing to obtain the power drain across the entire day.

Note that for the geofence regimes, we use the geofence coefficient for the passive and active states, and the selected sensing regime for the moving state.

This gives us the distribution of power drains across the set of users for each regime. A boxplot of these distributions is shown in Figure 5. Unsurprisingly, the `nohafs` regime runs out of battery for almost all users on both platforms. In order to get more visibility into the details of the other regimes, we re-plotted the graph after excluding `nohafs`. From that graph, we can observe that:

1. The graphs are actually fairly consistent across platforms - the median drain for all geofenced regimes on both platforms is around 20%.
2. The major difference between the geofenced regimes is in the spread of the data - the medium accuracy regimes with geofencing have tighter bounds than the `geo-hafs` regime, and fewer outliers. Some of

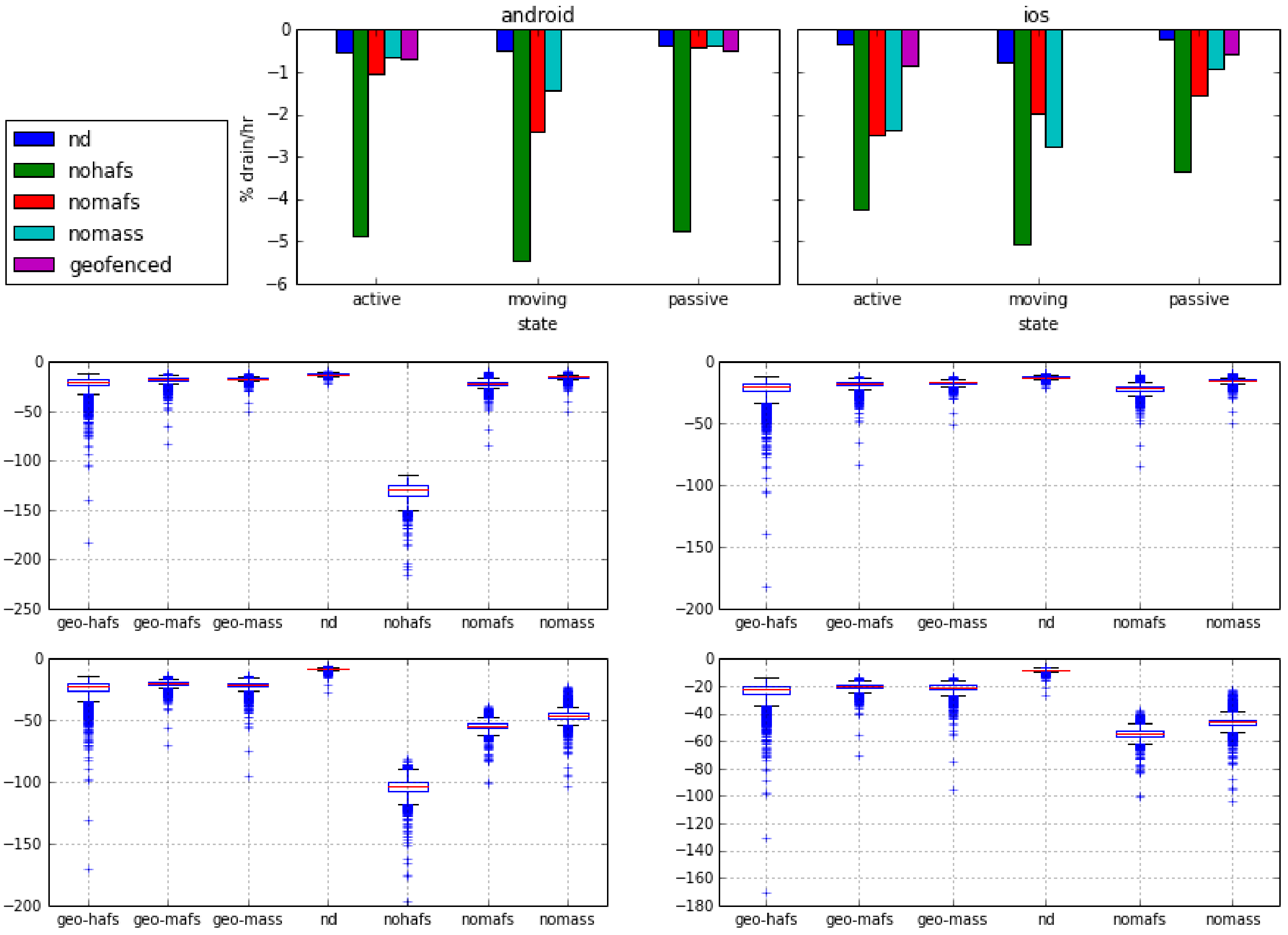

Figure 5: Results for generalizing the results to a broader variety of activity patterns. Top: a simple model for estimating the power drain as a factor of the no data collection `nd`, tracking using high accuracy fast sampling `nohafs`, tracking using medium accuracy fast sampling `nomafs`, tracking using medium accuracy slow sampling `nomass` and geofenced `geofenced` states. Bottom: Distributions of power drain (%/hr) in each regime generated by applying the top model to the ATUS dataset

the outliers in the `geo-hafs` case are greater than 100%, indicating that in a few cases, the phone will run out of battery even with geofencing turned on.

3. The non-geofenced regimes are noticeably different - the median on iOS is around 50%, while the median on Android is at the same (15 - 20%) as the other data collection methods. This implies that by using medium accuracy on Android, it might be possible to get away without duty cycling. But it is clear that for iOS, any reasonable data collection solution must use some form of duty cycling.

7. CONCLUSION AND FUTURE WORK

As we have seen in the boxplots in Section 5 the current data collection is skewed because we are unable to detect the end of a trip until the end of the hour when we receive a remote push. We should investigate other techniques for detecting a trip end. Some examples are to use the newly created Visit API, or to perform inverse duty collection in which we collect points with finer granularity as the speed reduces. This will effectively remove the distance filter as we come to a stop, and allow us to collect enough points to detect the trip end.

The results also show that medium accuracy data collection on Android has the same power drain as geofencing. This suggests the exploration of an alternate duty cycling method in which we switch to low accuracy sensing instead of geofencing in the active state.

In conclusion, the data that we have collected shows that passive sensing requires duty cycling to be practial. There are multiple duty cycling methods that can work on Android but using geofencing as the trigger for a cascade is the only feasible option in iOS. We have built an open source library for both Android and iOS that implements these techniques, and we are able to share some of the challenges that we encountered.

8. REFERENCES

[1] U. Bareth and A. Kupper. Energy-Efficient Position Tracking in Proactive Location-Based Services for Smartphone Environments. pages 516–521. IEEE, July 2011.

[2] 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 *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 54–67. ACM, 2011.

[3] J. Hood, E. Sall, and B. Charlton. A GPS-based bicycle route choice model for San Francisco, California. *Transportation Letters: The International Journal of Transportation Research*, 3(1):63–75, Jan. 2011.

[4] G. Inc. See and manage your timeline. `https://support.google.com/gmm/answer/6235133?hl=en`, 2015. [Online; accessed 09-Dec-2015].

[5] M. B. Kjærgaard, J. Langdal, T. Godsk, and T. Toftkjær. Entracked: energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 221–234. ACM, 2009.

[6] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 563–575. USENIX Association, 2015.

[7] S. Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 29–42. ACM, 2012.

[8] J. Paek, J. Kim, and R. Govindan. Energy-efficient rate-adaptive GPS-based positioning for smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 299–314. ACM, 2010.

[9] ProtoGeo. Moves on the App Store. `https://itunes.apple.com/us/app/moves/id509204969?mt=8`, 2013. [Online; accessed 09-Dec-2015].

[10] S. Reddy, K. Shilton, G. Denisov, C. Cenizal, D. Estrin, and M. Srivastava. Biketastic: sensing and mapping for better biking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1817–1820. ACM, 2010.

[11] K. Shankari. e-mission-data-collection repository. `https://github.com/e-mission/e-mission-data-collection`, 2015. [Online; accessed 09-Dec-2015].

[12] K. Shankari, M. Yin, D. Culler, and R. H. Katz. E-Mission: Automated transportation emission calculation using smartphones. In *Pervasive Computing and Communication Workshops (PerCom workshops)*, pages 268–271, Mar. 2015.

[13] V. Srinivasan and T. Phan. An accurate two-tier classifier for efficient duty-cycling of smartphone activity recognition systems. In *Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*, page 11, 2012.

[14] F. Viticci. The Background Data and Battery Usage of FacebookâĂŹs iOS App. `https://www.macstories.net/linked/the-background-data-and-battery-usage-of-facebooks-i` 2015. [Online; accessed 09-Dec-2015].

[15] Z. Yan, V. Subbaraju, D. Chakraborty, A. Misra, and K. Aberer. Energy-efficient continuous activity recognition on mobile phones: an activity-adaptive approach. In *Wearable Computers (ISWC), 2012 16th International Symposium on*, pages 17–24, 2012.

[16] M. Zhong, J. Wen, P. Hu, and J. Indulska. Advancing Android activity recognition service with Markov smoother. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*, pages 38–43. IEEE, 2015.