An Energy-efficient Location Provider for Daily Trips

Julien Duribreux, Romain Rouvoy, Martin Monperrus

An Energy-efficient Location Provider for Daily Trips

Julien Duribreux, Romain Rouvoy, Martin Monperrus

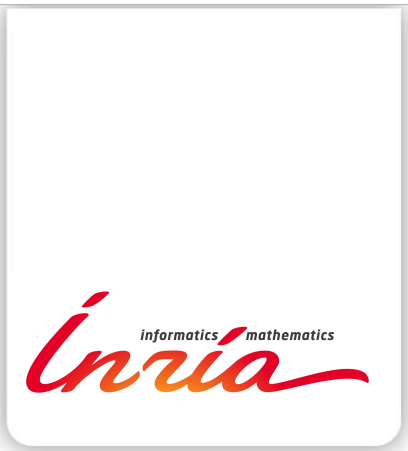
RESEARCH REPORT

N° 8586

August 2014

Team Spirals

ISSN 0249-6399 ISRN INRIA/RR--8586--FR+ENG



An Energy-efficient Location Provider for Daily Trips

Julien Duribreux, Romain Rouvoy, Martin Monperrus

Team Spirals

Research Report n° 8586 — August 2014 — 18 pages

Abstract: It is well known that smartphones are energy greedy and their batteries do not last more than a few days. Since mobile phones consumption increased faster than batteries capacities it becomes important to find new ways to reduce their energy consumption.
We claim that people using GPS location service everyday should not pay a full energy-cost since most of the time their are taking the same path to commute.
Based on these observations, we developed a new implementation of the location API built from Android Location API, which is capable of learning and predict user location on time with an average accuracy of 50 meters while saving energy.

Key-words: energy-efficient, smartphone, clustering, location inference

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Un GPS à faible coût énergétique pour les trajets quotidiens

Résumé : Il est généralement admis que les smartphones sont des appareils qui consomment énormément d'énergie et qui nécessitent d'être rechargés régulièrement. Cet état de fait est en partie lié à l'évolution de la consommation de ces derniers qui augmente plus rapidement que celle de la capacité des batteries.

Aussi, nous observons qu'une grande partie de la population utilise la localisation par GPS quotidiennement et en parcourant généralement les mêmes routes afin de, par exemple, connaître les conditions de circulation ou du temps restant.

À partir de ces observations, nous proposons une nouvelle mise en œuvre de l'API de localisation, basée sur celle d'Android, capable d'apprendre et de prédire la position des utilisateurs avec une précision moyenne de cinquante mètres tout en diminuant sa consommation au fil du temps.

Mots-clés : Energie, smartphone, GPS, localisation

Contents

1 Introduction **4**

2 State of the Art **4**
2.1 Energy-consumtion Understanding 4
2.2 Offloading Strategies 5
2.3 Delaying Strategies 6
2.4 Code Modification Strategies 6

3 Materials and Methods **7**
3.1 Hardware Workbench 7
3.1.1 Hardware Setup 7
3.1.2 Why Using Cyanogen? 7
3.2 Energy Usage Capture 8
3.3 Motivations for a Smart Location API 8

4 Algorithm Description **9**
4.1 Learning Phase 9
4.2 Prediction Phase 11
4.2.1 Path Selection 11
4.2.2 Mobility Correction 12
4.3 Dalvik Bytecode Modification 12

5 Results **13**
5.1 Consumption Analysis 14
5.1.1 Learning Phase 14
5.1.2 Prediction Phase 14
5.2 Threats to Validity 16

6 Conclusion **16**

1 Introduction

It is not a secret, smartphones are taking a bigger place in our society and manufacturers tend to increase their sizes and power everyday. Because of that, smartphones can only stay unplugged one or two days in a raw without us having to refill their batteries. However, autonomy is one of the most important and challenging user requirement. To face that reality, we need to constantly improve energy policy in smartphones to increase the battery lifetime.

Additionnally, we observe that some applications tend to drain batteries faster than the others. New bugs appear— so called energy-bugs [18]—and one cannot directly notive them because applications are working perfectly, but behind the scene these bugs are draining the battery. For example, no-sleep bugs are defined as energy bugs resulting from mis-handling power control APIs in an app or framework, resulting in the smartphone components (screen, cpu, gps...) staying on for an unnecessarily long period of time [17], which is a big deal considering we already have limited autonomy. To improve the battery lifetime, we therefore have to develop and deploy smarter algorithms able to adapt application behavior to reduce energy cost.

Li et al. [7] show that a major cause of energy leaks in mobile applications are communication and screen display, CPU is no more that greedy thanks to frequency scaling policy used in recent mobiles. On our side, we decided to focus on the GPS knowing that a typical smartphone will completely drain its battery in about 6 hours if the GPS is running continuously. Note that most of the energy is used in computing the location, not retrieving data from satellites [12] since there is no communication from the smartphone to satellites.

The key contributions of this paper are therefore:

1. A state of the art of energy-efficient software-based optimizations on mobile phones,
2. A low cost technique to measure mobile phones energy consumption at run-time,
3. An energy-efficient location computing process based on machine learning.

2 State of the Art

Many researchers all around the world are working on the energy challenge for mobile devices to figure out new ways to reduce their hardware and applications consumption. In this paper, we are focusing on software-based solutions because it does not require any hardware manipulations or Android SDK modifications and we think there is more potential for real-world deployment.

2.1 Energy-consumtion Understanding

Saving energy is only possible with good understanding of smartphones and applications behaviors. In this part we are focusing on tools and analysis toward a better understanding of smartphones.

Pathak et al. present the first comprehensive real-world no-sleep energy bug characterization study, then they also proposes a detection solution based on the classic reaching definitions dataflow analysis, to automatically infer potential no-sleep bugs in an app [18]. This is a good first step to understand how does the energy consumption model works in mobile phones and helps developers to build great energy-efficient applications.

Bavota et al. investigate energy-greedy Android APIs, with the purpose of understanding particular instances of API calls and API usage patterns that cause high energy consumption [4]. However, they were only interested in patterns composed of calls to public Android methods. For this reason, they removed Android non public methods (private and protected methods) from the identified patterns. In particular, analysis shows that APIs related to *GUI and Image Manipulation and Database represent, all together, 60% of the energy- greedy APIs.*

Li et al. present a tool able to compute source line level energy consumption information combining hardware-based power measurements with program analysis and statistical modeling [11]. Their approach is fast, it can calculate source line level energy measurements for each of their subject applications in less than 3 minutes and is able to calculate energy consumption within 10% of the ground truth measurements.

2.2 Offloading Strategies

Those strategies are based on the fact that it may take time to achieve heavy computing operations. Thus, it might be interesting to offload those calculus on a remote server and get final results. Even if sending data will increase consumption at some point, it is possible to achieve energy saving at application level because of the heavy computations.

Aggarwal et al. observe that *radio communications are a significant cause of battery energy drain* in smartphones. They tried to reduce this cost by using aggregation, instead of sending request one by one, they merge requests into communication spurs to reduce the number of state switches (low and high power mod), or using data compression with a cloud proxy to compress data before sending it. To this end, they employ an asymmetric, lossless, dictionary-based redundancy elimination technique, with minimal decompression overhead. Thus, bandwidth savings translate into equivalent energy savings. And finally, using scheduling, they send data based on signal variations [1]. Overall this technique suffers from its very specific behavior because it relies on a distant server able to (de)compress data which is non-standard and would probably be difficult to deploy at large.

Kwon et al. use annotation to automatically transforms the application to enable it to offload parts of its functionality to the cloud. Doing that, they can effectively reduce the overall amount of energy consumed by these applications, with the actual numbers ranging between *25% and 50%*. To do so, the offloading decisions should be made dynamically at runtime and continuously adjusted in response to the fluctuations in the mobile execution environment. Depending on the runtime execution environment, different portions of a program's state can be checkpointed and transferred across the network as required by the offloading strategy in place. To select the optimal offloading strategies, they combine dataflow and side-effect analyses. Based on the results, a bytecode enhancer then rewrites the application without changing its source code. In particular, they used Soot to implement their program analysis and transformations [8].

Merrer et al. created Hoop, a tool based on the fact that it takes time to upload HD pictures, videos and other heavy media on Internet. The idea is quite straightforward: instead of directly uploading the file to the online service, the user browser encrypts and uploads the file to the gateway, together with an authentication token, at a speed determined by the Wi-Fi connection of the access point. At this point, the user can disconnect from the access point, and potentially move and switch off her device, while the file is being asynchronously uploaded by the gateway. The offload time is *reduced by up to a factor of 85 in a wired setting and by up to a factor of 46*

in the wireless settings [14]. However, their optimization relies on custom gateway that may not be available everywhere and even if there is a gain in time, there is no energy saving since the gateway does the job.

2.3 Delaying Strategies

Those strategies focus on the fact that sending data as soon as possible may not be the best choice because of the components energy tail state. If the application is delay-tolerant, delay some operations avoid useless tail state and doing so, save energy.

Wang et al. try to find out how the energy consumption characteristics vary depending on the packet size and transmission interval, and also how to minimize the power consumption in case of constant bit rate data transmission [21]. They proposed a power consumption model based on their experimental results, taking into account packet sending intervals and packet size. Their results show that proposed parameters selection allows to significantly decrease the uplink power consumption on the mobile device taking signalling traffic and latency restrictions into account.

Balasubramanian et al. compare energy consumption of different mobile networking technologies. Long story short, it appears that WiFi is the most efficient way to exchange data once we are connected to an access point (AP). Otherwise, **2G** is more efficient than **3G** except for big transfert and strong signal. Although, this is interesting to understand that WiFi searching for AP consumes a lot of energy. They also introduce *TailEnder* that aggressively prefetches data, including potentially useless data, and yet reduces the overall energy usage. Delaying data sending can reduce energy by *35% for email applications, 52% for news feeds, and 40% for web search* [2]. Even if this is a powerful strategy, this is only working on time tolerant applications, which cannot always be accepted for mobile apps.

Nikzad et al. create an annotation language (APE) to demarcate a power-hungry code segment whose execution is deferred until the device enters a state that minimizes the cost of that operation [16]. Empirical evaluation shows that APE introduces an negligible overhead and achieve *63.4%* energy savings compared to the case when there is no coordination between applications to save up energy.

2.4 Code Modification Strategies

The main idea here is that original source code may not be energy-efficient. To improve their behaviour, those strategies generate some alternatives versions of the same application searching for the most energy-efficient.

Manotas et al. create *SEEDSapi* [13]. It generates application versions implementing many different alternative combinations of API implementation choices for all the object instantiation locations in the application. It performs power-monitored executions for a given test suite on all the generated versions, analyzes the collected energy usage data to identify the best combination of API implementation choices per object allocation location, and generates an optimized version of the application based on API implementation decision-making. SEEDSapi optimizes Java applications by identifying implementations of the Java Collections API that are more energy efficient by directly modifying the application's bytecode, if any, than the implementations currently used by the application. SEEDSapi improved the energy usage of subject applications by between 2 and 17%.

Li et al. propose a new technique for automatically transforming the color scheme of a mobile web application [10]. The approach rewrites the server-side code and templates of a web application so that the resulting web application generates pages that are more energy efficient when displayed on a smartphone. In the evaluation of their approach, they showed that they can achieve a 40 percents reduction in display power consumption. A user study indicates that the transformed web pages are acceptable to users with over 60 percents indicated that the transformed version would be acceptable for general use given the energy savings, and over 97 percents said it would be acceptable for use if the battery power was critically low.

Thokala et al. propose Virtual GPS, a middleware layer that provides current location to the applications in a power efficient manner using a cross-layer approach [20]. The main contribution of their work is energy efficiency in determining the location by considering accuracy needed of applications and dynamically determining sensor type using. To do so, they use multiple techniques depending on that precision is required. Using the Virtual GPS layer to abstract location sensing provides solution that could save around 30% of energy when compared to existing solutions.

Synthesis A lot of work as already been done in this field, but a lot more has to be done if we want to keep increasing size and power of smartphones. In our study, we are focusing on location computing because we think that we can achieve major energy savings.

3 Materials and Methods

3.1 Hardware Workbench

In our experiment we use a Samsung I9300 Galaxy S III, which is a recent and powerful smartphone running on Android 4.3.1 with a custom CyanogenMod firmware [6], and we capture energy consumption with a java application running on a MacBook Pro.

3.1.1 Hardware Setup

To plug our ammeter, we need to pull out the battery, wire our device between the positive out of the battery and the positive pin inside the smartphone. Then, we link both minus together. Next, we plug the ammeter with mini-usb to a computer and it is ready to run (cf. Figure 1). Notice that we cannot recharge our battery inside the smartphone now, to avoid having to unmount everything we use a direct current lab generator configured to deliver 5V at 0.6mA. We could go to 1.2mA but batteries do not like fast-recharge, so to preserve it, we reduce the output current.

3.1.2 Why Using Cyanogen?

The Samsung I9300 battery has 4 small pins to communicate and transfer energy to the smartphone. Due to lack of room to link all of them together, we only focused on the significant ones, which means the positive and negative power pins. Because of that the Samsung front-end was able to detect a malfunction on the battery and keep dropping pop-ups warning to us about that. Obviously, this is not a kind of behavior we want because it may put the application running on foreground to the background and changing the energy consumption of the smartphone.

Furthermore, Samsung front-end adds functionalities, which is great for a friendly user interface, but behind the scene it may cause energy overhead while running non-critical services. Changing to Cyanogen ensures us to have a clean install on the smartphone with minimum exotic

extras running. Note that we also use the smartphone airplane mode to avoid energy noise from other components like WiFi, 3G...

3.2 Energy Usage Capture

To be able to quantify energy gain from our optimizations, we need to measure precisely the smartphone energy consumption. To do so, there are two major techniques. The first one is to estimate energy usage using energy models based on hardware characteristics with tools like PowerTutor [19]. Overall this technique can suffer from low precision and induce energy consumption overhead due to the energy estimation process. And, of course, every smartphone is different, which means that if we want to catch energy consumption on a different devices we have to re-configure the energy model with parameters like screen size, battery capacity, CPU frequency scaling... With the second one, we can measure energy usage of the physical device, which gives us a great precision, but tools like Monsoon power meter [15] use to be very expensive and it requires some basic knowledge in circuitry. To avoid these limitations, we use Yocto-Amp from Yoctopuce [22]. This device is a digital USB ammeter that allows us to measure current automatically. It can provide rather accurate digital measures (2 mA, 1%) and it is very easy to use because of its great API avaible in 10 languages, giving us the possibility to do whatever we want with the ammeter input data.

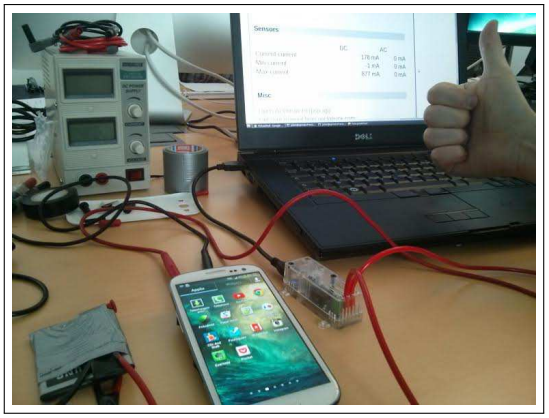


Figure 1: Galaxy S3 plugged to the ammeter and running.

3.3 Motivations for a Smart Location API

To achieve energy saving we decided to focus on software-based solution because we can use them with most of the applications on the Google Play Store. Indeed, modifying Android SDK or the smartphone hardware would require a very specific configuration to run our optimization and so, would decrease our potential impact on real world devices running on various Android versions and hardware.

We claim that people using GPS location service daily should not pay a full energy-cost since most of the time their are taking the exact same path, therefore we should be able interpolate their position in real time based on traveling patterns analysis and save battery lifetime.

To achieve energy saving, we try to reduce the number of GPS calls and avoid tail state statuses by learning commuting road trips and then, the next time user takes the same road, instead of calling for a GPS fix our API will compute what should be the next location based on

previous location information. Doing so, we are trying to avoid energy tail state, which is a well known phenomenon causing energy leaks [9] and save up energy used to compute user location (cf. Figure 2).

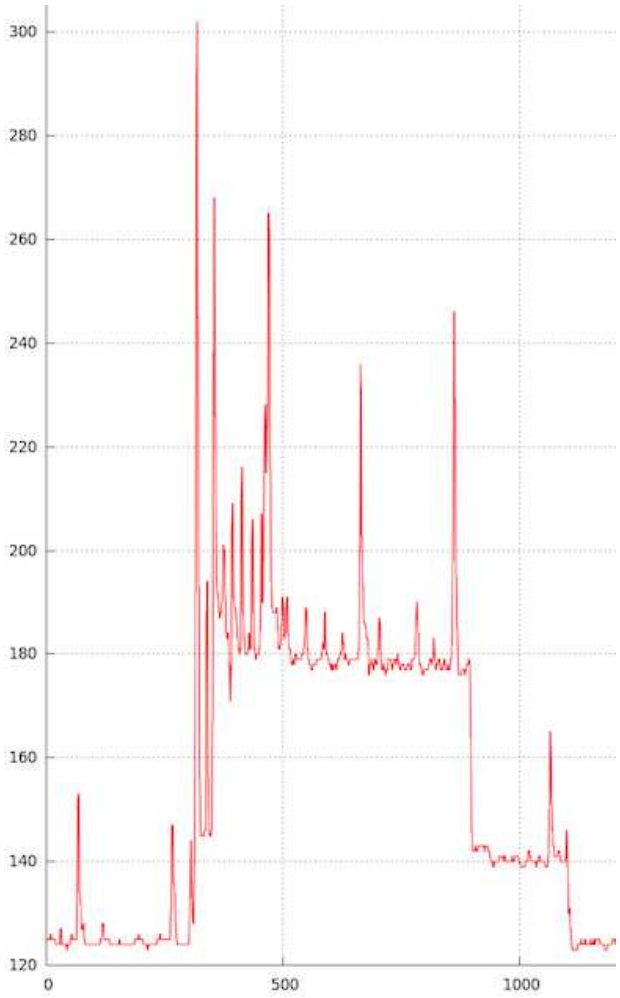


Figure 2: GPS tail state visualization in mA during a GPS fix from Android API captured with Yocto-ammeter.

4 Algorithm Description

4.1 Learning Phase

To expect being able to estimate user location, we first need to learn its daily travelling patterns using GPS. Every GPS fix is what we call a **GPSNode** and **Clusters** are built with them (cf. Figure 3).

First During this phase, we need to collect as many locations as possible like any navigation application does which means continuously request for GPS fixes. Of course we do not

Figure 3: Clusters on a Google Map. Colors indicate the direction where the user is going.

Data: F is the frequency of location requests, M is our prediction model containing the clusters list
$T \leftarrow 0$; `// Counter`
$O \leftarrow null$; `// Location prediction`
$R \leftarrow 11$; `// Predict ratio, 1 per 11`
while *App is running* **do**
  get real location L with Android API;
  stop listening for real GPS fix;
  get closest Cluster C from L;
  if L *is in max range from* C **then**
    Update C with L;
    Update transition from C to O;
    $R \leftarrow R + 1$;
  else
    Create a new Cluster C with L;
    Create a transition from O to C;
    Add C to our Clusters list;
    Reset R;
  end
  $0 \leftarrow L$;
  Return L;
end

Algorithm 1: Learning algorithm — M

expect that this kind of behavior save any energy considering it is the same behavior as usual;

Second Every time we get a fix, we store its information in a SQLite database, which means latitude, longitude but also speed, bearing, altitude and timestamp. Plus, we ignore information coming from fixes with accuracy below 15 meters to insure a decent accuracy while learning the trip;

Third For each new GPSNode, we check the distance from the closest cluster, if it is the same cluster as before, the user did not move, so we decide to ignore this fix. If it is within our range limit (15 meters), we update its information doing arithmetic means. If it is not the case, we create a new cluster with GPSNode information.

Fourth In addition, we also need to know how long did it take to move from cluster to cluster. To do so, we need another relation which contains time between them and the orientation, as going from A to B is not the same than B to A (cf. Figures 4 and 5). Note that we only consider the time between clusters, the time spent inside a cluster does not count and we fixed minimal distance between clusters to 15 meters.

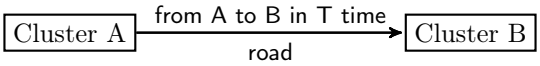


Figure 4: Relations between clusters in our SQLite Dabase

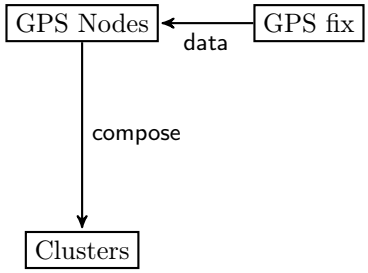


Figure 5: Relations between GPSNodes and Clusters in our SQLite Database

4.2 Prediction Phase

Once we got enough Clusters our custom `LocationManager` is able to estimate current user location using time elapsed between last prediction. We fix the minimum duration between GPS calls to 11 seconds to avoid energy waste in tail states, but every second we predict what should be the actual user location like a real GPS does. When a real gps fix occurs, we compare our estimated location to the real one and decide or not to increase time between calls depending on how precise we are.

4.2.1 Path Selection

It may happen that some parts of the road are shared by different trips and at some point the user decides to go for one or the other. In those cases, it is important to make the most probable

```
while App is running and every F seconds do
    T ← T + 1;
    if T mod R equals 0 then
        Request for real GPS fix;
        Return our prediction;
    else
        Compute static time S;
        Compute prediction P knowing O and time elapsed F − S;
        if we cannot make a prediction then
            Return what I am learning on M;
        else
            0 ← P;
            Return P;
        end
    end
end
```

Algorithm 2: Prediction algorithm

direction. In our experiment, we are considering how many times both tracks have been chosen and go for the most chosen. Sometimes, it may not be enough, but other techniques should be considered: using average timestamp and make the decision on what time it is instead of the number of occurrence.

4.2.2 Mobility Correction

Obviously, road traffic may not be the same, so time between clusters may change and fails the prediction in a big way if the car stops, for example. To avoid this kind of problem, we are using the accelerometer inside the mobile phone to know if we are moving or not. It should be apparent that in order to measure the real acceleration of the device, the contribution of the force of gravity must be eliminated. This can be achieved by applying a high-pass filter. Then, we make the sum of all acceleration axis and compute the absolute value.

$$\sigma = Math.abs(\alpha + \beta + \gamma)$$

Empirical study showed that sigma value must be of 0.25 (cf. Figure 6). Under this critical point, the car is not moving and above it is. It is important to notice that, due to the vibrations of the car and high sensors sensitivity, the value cannot be 0. Plus, even if we are moving at constant speed, which means acceleration should be close to the non-moving one, we are taking benefits of the variations of the road like holes, surface contact which increase vibrations amplitude to make make the difference between those two movement states. Also it may happen that in very rares circumstances, we got short periods of time while σ is below our limit. Due to the `getTimestamp` method accuracy 1ms may be changed in 100ms, to avoid false positive, we only arbitrary consider period of stability above 500ms.

4.3 Dalvik Bytecode Modification

Due to lack of time, this part is not fully explored, but preliminary results are encouraging. To be able to easily use our API inside another application, we decide to match almost perfectly the

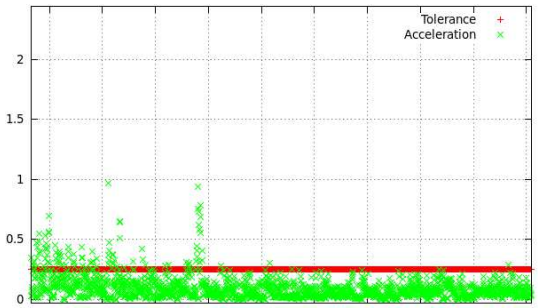


Figure 6: Variation of the acceleration during time. At first, the car is moving then it stops and finally there is no movement, the car is static. The red line is our limit at 0.25

Android `LocationManager` class, we only need one more thing, which is the application `Context` in our custom `LocationManager` constructor (cf. Listing 1).

Listing 1: Call for location update

```
manager = new MyLocationManager(getApplicationContext());
manager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
    mLocationManager);
```

To use the new location API in existing applications, without having source code, we need to work on the Dalvik Bytecode. To do so, we are using Dexpler [3], based on Soot, which is a language manipulation and optimization framework. Since our modification is hidden behind our custom `LocationManager`, we just have to find and modify the Android `LocationManager` objects inside the targeted application and change it with our optimized one. When the application requests for location, instead of returning the last known location we got from real GPS fix using `getLastKnownLocation` method from the Android Location API, we will give away our last prediction computed by our prediction component.

About this technique :

Pro We can modify any application even if we do not own the source code.

Con Modifying APKs forced us to resign the application with our personal key. Doing so the application signature would be the same as the original and we will no longer benefit from future updates.

5 Results

This section presents an evaluation of our approach. First, we discuss the consumption overhead induce by our custom `LocationManager` during the learning phase. Then, we compare the energy consumption during the prediction and analyse how accurate our predictions are. Our optimization has been tested on the same application, running the Android Location Provider and our custom Location Provider during a road trip of 2.5km and 3.2km.

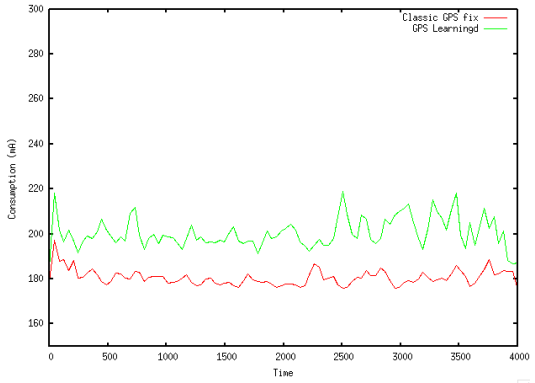


Figure 7: Consumption comparison between a classic GPS and our GPS during the learning phase with smooth bezier drawing

5.1 Consumption Analysis

5.1.1 Learning Phase

While learning, we can hardly expect any energy saving because we need at first to capture user trips and to do so, request for GPS fixes continuously and store all those data. Plus, even if there is no prediction yet, our prediction component is already running, trying to find nearby cluster to compute the next location.

As expected, *learning creates an energy overhead of about 11%* (cf. Figure 7), which seems acceptable considering GPS will drain the battery in only 6 hours, it represents about 40 minutes loss. Overall, this only occurs during roads discovery, as once the road is known, the prediction component will reduce the number of GPS calls and hopefully save up energy.

5.1.2 Prediction Phase

Once the road is well known, the number of GPS calls will start to decrease progressively starting at 1 every 11 seconds and, if predictions are good enough, it will keep going down. This choice is motivated by the duration of the tail state (cf. Figure 2). After 11 seconds, the period of heavy consumption ends and only few seconds of low consumption remains, +8% compared to the idle state. Plus, we want to avoid long period of no location verification because our prediction may go wrong and sending wrong location at lost energy-cost is not what we want to achieve.

Despite our best efforts, the application struggles to achieve energy saving. Even if at some points global consumption drops down under the regular GPS consumption, the computing overhead counter our energy gain (cf. Figure 8). Our average consumption is 190mA, which is slightly above the classic GPS behavior at 180mA. Overall, this is only 5.5% more and the cause of this result is that we are still requesting for real GPS fix too often. This behavior is due to our reset constraints: if we are unable to find a cluster in a close range or if our prediction is too bad, above 30 meters away from real location, then we try to improve our model by searching for a GPS fix. This decision may not be bad, but only works when we have a good learning model, at the beginning it cannot be energy-efficient.

The average accuracy is about 47 meters (cf. Figure 9), which is quite good considering the GPS checking interval and variation due to environmental factors. This suggests that we maybe could achieve energy saving using this technique by forcing GPS fixes every 15 seconds or more and keep a prediction good accuracy.

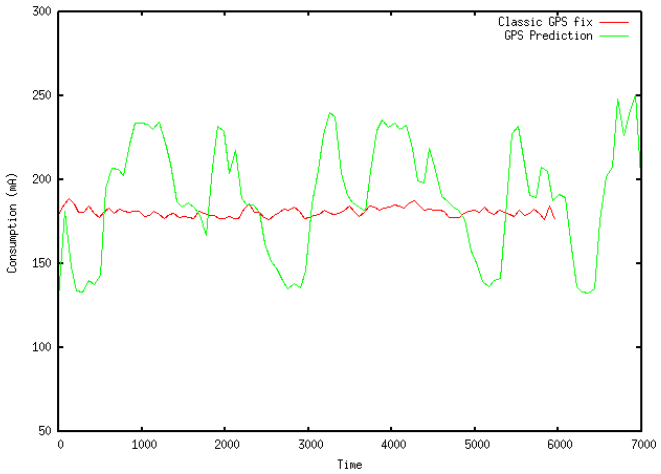


Figure 8: Consumption evolution during prediction phase compared to default GPS behavior

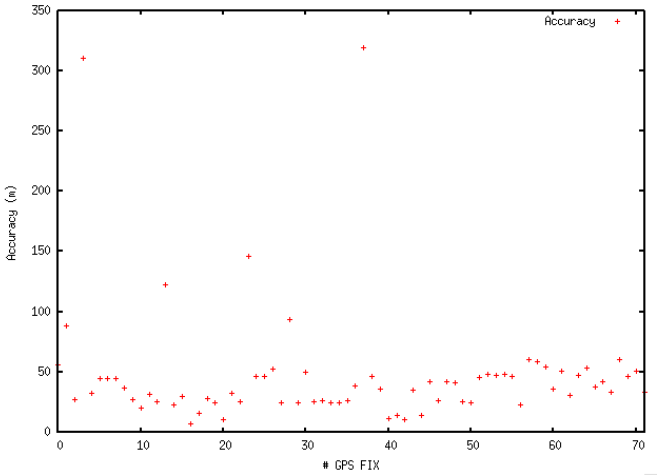


Figure 9: Evolution of accuracy during prediction phase

5.2 Threats to Validity

Obviously, our model is not perfect and prediction may not be accurate due to multiple reasons.

First Learning conditions may vary in a huge way due to environmental factors. One of the major cause is the weather: if the sky is clear, we can expect excellent location accuracy, more or less 5 meters. On the other hand, if the sky is cloudy, accuracy will drop significantly to 15 meters or more. This is due to difficulties to acquire GPS satellites, variation in the signal quality and lag;

Second Even if we are trying to reduce its effect, traffic can create time variations between clusters, which is crucial for our prediction. For example, *10km/h* variation is equal to **2.77m/s**. Thus, if the prediction occurs every 11 seconds (which is the default time we use to avoid energy tail state and hope for energy saving), it could be **30m** wrong from the actual location. At the moment we are unable to estimate speed based on accelerometer due to its instability, so we cannot adjust our prediction based on deceleration or acceleration;

Third Due to the lack of data on some trips, our prediction model may not be able to compute user location. If the closest cluster is too far away from the current location our custom location provider will request for GPS fix to correct its lack of knowledge and doing so it will increase its global energy consumption and return a bad prediction.

6 Conclusion

In this paper, we propose a new Location Provider API based on Android Location API able to estimate user's location within 50 meters of accuracy and able to save energy based on daily learning. Compared to previous researches on smartphone energy-saving our API can be used in any existent applications using Dexpler to modify Dalvik bytecode and generate optimized APKs.

We believe that real-time location approximations based on daily trips can still be improved using smartphones sensors. For instance, with better usage of the accelerometer it may be possible to compute the user speed. With this information, it should be possible estimate user location accurately using time and speed.

In the same spirit, compass do not require a lot of energy and can be use to detect orientation decision. Knowing that and the speed, it should theoretically be possible to compute user location with basic maths and only one real GPS fix.

Finally, sharing current location between different smartphones close to each others using Bluetooth Low Energy [5] should be an interesting way to share or compute user location at low energy cost knowing that the peak current consumption is only less than 15mA. Sharing the information with peers should be more efficient than computing the same location many times.

References

[1] Bhavish Aggarwal et al. "Stratus: Energy-Efficient Mobile Communication using Cloud Support". In: (2010), pp. 477–478.

[2] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications". In: (2009).

[3] Alexandre Bartel et al. "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot". In: *. . . Art in Java Program analysis* Soap (2012), pp. 27–38. URL: http://doi.acm.org/10.1145/2259051.2259056.

[4] Gabriele Bavota et al. "Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study". In: ().

[5] *Bluetooth LE*. URL: http://www.wikiwand.com/en/Bluetooth_low_energy.

[6] *Cyanogen*. URL: http://wiki.cyanogenmod.org/w/About#About_CyanogenMod.

[7] Jiaping Gui Ding Li Shuai Hao. "An Empirical Study of the Energy Consumption of Android Applications". In: (2014).

[8] Young-Woo Kwon and Eli Tilevich. "Reducing the Energy Consumption of Mobile Applications Behind the Scenes". In: *2013 IEEE International Conference on Software Maintenance* (Sept. 2013), pp. 170–179. DOI: 10.1109/ICSM.2013.28. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6676888.

[9] Nicholas D Lane et al. "Piggyback CrowdSensing (PCS): Energy Efficient Crowdsourcing of Mobile Sensor Data by Exploiting Smartphone App Opportunities". In: ().

[10] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. "Making web applications more energy efficient for OLED smartphones". In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014), pp. 527–538. DOI: 10.1145/2568225.2568321. URL: http://dl.acm.org/citation.cfm?doid=2568225.2568321.

[11] Ding Li et al. "Calculating source line level energy information for Android applications". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013* (2013), p. 78. DOI: 10.1145/2483760.2483780. URL: http://dl.acm.org/citation.cfm?doid=2483760.2483780.

[12] Jie Liu et al. "Energy efficient GPS sensing with cloud offloading". In: *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems - SenSys '12* (2012), p. 85. DOI: 10.1145/2426656.2426666. URL: http://dl.acm.org/citation.cfm?doid=2426656.2426666.

[13] Irene Manotas, Lori Pollock, and James Clause. "SEEDS: a software engineer's energy-optimization decision support framework". In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014), pp. 503–514. DOI: 10.1145/2568225.2568297. URL: http://dl.acm.org/citation.cfm?doid=2568225.2568297.

[14] Erwan Le Merrer, Nicolas Le Scouarnec, and Gilles Straub. "Hoop: Offloading HTTP(S) POSTs from User Devices onto Residential Gateways". In: ().

[15] *Monsoon*. URL: https://www.msoon.com/LabEquipment/PowerMonitor.

[16] Nima Nikzad, Octav Chipara, and William G Griswold. "APE: An Annotation Language and Middleware for Energy-Efficient Mobile Application Development". In: (2014).

[17] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. "Where is the energy spent inside my app?" In: *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12* (2012), p. 29. DOI: 10.1145/2168836.2168841. URL: http://dl.acm.org/citation.cfm?doid=2168836.2168841.

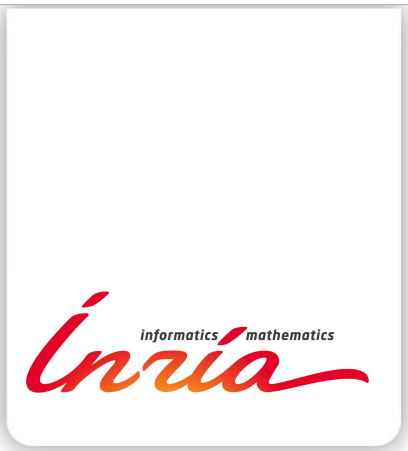
[18] Abhinav Pathak et al. "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps". In: *. . . on Mobile systems, applications, . . .* (2012), pp. 267–280. URL: http://dl.acm.org/citation.cfm?id=2307661.

[19] *PowerTutor*. URL: http://powertutor.org..

[20] Sravan Kumar Thokala. "Virtual GPS: A Middleware for Power Efficient Localization of Smartphones Using Cross Layer Approach." In: ().

[21] Le Wang, Anna Ukhanova, and Evgeny Belyaev. "Power consumption analysis of constant bit rate data transmission over 3G mobile wireless networks". In: *2011 11th International Conference on ITS Telecommunications* (Aug. 2011), pp. 217–223. DOI: 10.1109/ITST.2011.6060056. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6060056.

[22] *YoctoPuce*. URL: http://www.yoctopuce.com.



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399