View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Detecting Anomalous Energy Consumption in Android Applications

Tiago Carção, Marco Couto, Jácome Cunha,
João Paulo Fernandes, and João Saraiva

HASLab / INESC TEC, Universidade do Minho, Portugal
CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
RELEASE, Universidade da Beira Interior, Portugal
{tiagocarcao,mcouto,jacome,jpaulo,jas}@di.uminho.pt

Abstract. The use of powerful mobile devices, like smartphones, tablets and laptops, are changing the way programmers develop software. While in the past the primary goal to optimize software was the run time optimization, nowadays there is a growing awareness of the need to reduce energy consumption.
This paper presents a technique and a tool to detect anomalous energy consumption in Android applications, and to relate it directly with the source code of the application. We propose a dynamically calibrated model for energy consumption for the Android ecosystem, and that supports different devices. The model is then used as an API to monitor the application execution: first, we instrument the application source code so that we can relate energy consumption to the application source code; second, we use a statistical approach, based on fault-localization techniques, to localize abnormal energy consumption in the source code.

Keywords: Green Computing, Energy-aware Software, Source Code Analysis

1 Introduction

The software engineering and programming language communities have developed advanced and widely-used techniques to improve both programming productivity and program performance. For example, they developed powerful type and modular systems, model-driven software development approaches, integrated development environments, etc that, indeed, improve programming productivity. These communities are also concerned in providing efficient execution models for such programs, by using compiler-specific optimizations (like, tail recursion elimination), partial evaluation [1], incremental computation [2], just-in-time compilation [3], deforestation and strictification functional programs [4–6], etc. Most of those techniques aim at improving performance by reducing both execution time and memory consumption.

While in the previous century computer users and, as a consequence, software developers, were mainly looking for fast computer software, this is being chang-

ing with the advent of powerful mobile devices, like laptops, tablets and smartphones. In our mobile-device age, one of the main hardware/software bottlenecks is energy-consumption! In fact, mobile-device manufacturers and their users are as concerned about performance of their device as in battery consumption/lifetime! Unfortunately, developing energy-aware software is a difficult task, still. While programming languages provide several compiler optimizations, memory profiler tools, benchmark and time execution monitoring frameworks, there are no equivalent tools/frameworks to profile/optimize energy consumption.

In this paper we propose a methodology to monitor and detect anomalous energy consumption for the Android ecosystem: a widely used ecosystem for mobile devices. More precisely, we aim at providing Android application developers the tool support needed to develop energy-efficient applications. We propose a three layer methodology, which also account as the contributions of this paper:

- Firstly, we introduce an algorithm/application for the dynamic calibration of such model, thus allowing the automatic calibration of the model for any Android device. Moreover, we provide an API so that the calibrated power model can be accessed by the Android application we wish to monitor in terms of energy consumption.
- Secondly, we develop an Android application that automatically instruments the source code of a given Android application the developer wishes to monitor its energy consumption. The instrumentation is performed by embedding in the source code calls to the (calibrated) power consumption model API.
- Thirdly, we use a testing framework for Android applications in order to execute the (previously compiled) instrumented application. For each execution of a test case, we collect the energy consumed. Based on the energy consumed logs we performed several static analysis to detect abnormal energy consumption.

We have implemented our methodology in two different tools: one to dynamically calibrate our Android power consumption model, using a pre-defined set of calibrating applications. The second tool is used to automatically instrument the source code of an application its developed wishes to monitor in terms of energy consumption.

This paper is organized as follows: Section 2 presents the Android power consumption model, its dynamically calibrating algorithm, and the API that makes such model a reusable API. Section 3 describes the changes made to the Android power consumption model so it can be used to monitor power consumption at the source code level, as well as the changes that the framework does to an application source code. Section 4 introduces the framework GreenDroid, describing how it works and how it relates the previous sections. Section 5 describes the results generated by the framework. Finally sections 6 and 7 present the related work and the conclusions, respectively.

2 A Dynamic Power Consumption Model

In this Section we briefly discuss the Android power consumption model presented in [7]. This is a statically calibrated model that considers the energy consumption of the main hardware components of a mobile device. Next, we extend this model in two ways: firstly, we present an algorithm for thee automatic calibration of the model, so that it can be automatically ported to any Android based device (Section 2.2). Secondly, we provide an API-based implementation of the model so that it can be reused to monitor other Android applications (Section 3.1).

2.1 The Android Power Tutor Consumption Model

We know that different hardware components have different impact in a mobile device power consumption. As a consequence, an energy consumption model needs not only to consider the main hardware components of the device, but also its characteristics. Mobile devices are not different from other computer devices: they use different hardware components and computer architectures that have complete different impact in energy consumption. If we consider the CPU, different mobile devices can use very different CPU architectures (not only varying in computing power, but also, for example, in the number of CPU cores), that can also run at different frequencies. The Android ecosystem was design to support all different mobile (and non-mobile) devices (ranging from smart-watches to TVs). As a result, a power consumption model for Android needs to consider all the main hardware components and their different states (for example, CPU frequency, percentage of use, etc).

There are several power consumption models for the Android ecosystem [8–11, 7], that use the hardware characteristics of the device and possible states to provide a power model. Next, we briefly present the power tutor model [7]: a state-of-the-art power model for smartphones [8]. The Power Tutor [7] model currently considers six different hardware components: Display, CPU, GPS, Wi-Fi, 3G and Audio, and different states of such components, as described next.

CPU : CPU power consumption is strongly influenced by its use and frequency. The processor may run at different frequencies when it is needed, and depending on what is being done the percentage of utilization can vary between 1 and 100; There is a different coefficient of consumption for each frequency available on the processor. The consumption of this component at a specific time is calculated by multiplying the coefficient associated with the frequency in use with the percentage of utilization.

LCD : The LCD display power model considers only one state: the brightness. There is only one coefficient to be multiplied by the actual brightness level (that has 10 different levels).

GPS : This component of the power model depends on its mode (active, sleep or off). The number of available satellites or signal strength end up having little dependence on the power consumption, so the model has two power coefficients: one to use if the mode is on and another to use if the mode is sleep.

Wi-Fi : The Wi-Fi interface has four states: *low-power*, *high-power*, *low-transmit* and *high-transmit* (the last two are states that the network briefly enters when transmitting data). If the state of the Wi-Fi interface is *low-power*, the power consumption is constant (coefficient for *low-power* state), but if the state is *high-power* the power consumption depends on the number of packets transmitted/received, the uplink data rate and the uplink channel rate. The coefficient for this state is calculated taking this into account.

3G : This component of the model depends on the state it is operating, a little like the Wi-Fi component. The states are *CELL_DCH*, *CELL_FACH* and *IDLE*. The transition between states depends on data to transmit/receive and the inactivity time when in one state. There is a power coefficient for each of the states.

Audio : The audio is modeled my measuring the power consumption when not in use and when an audio file was playing at different volume, but the measures indicate that the volume does not interfere with the consumption, so it was neglected. There is only one coefficient to take into account if the audio interface is being used.

Static Model Calibration In order to determine the power consumption of each Android device's component the power model needs to be "exercised". That is to say, we need to execute programs that vary the variables of each components state (for example, by setting CPU utilization to highest and lowest values, or by configuring GPS state to extreme values by controlling activity and visibility of GPS satellites), while measuring the energy consumption of the device. By measuring the power consumption while varying the state of a component, it's possible to determine the values (coefficients) to include in a device's specific instantiation of the model.

Power Tutor, as all other similar power models, uses a static model calibration approach: the programs are executed in a specific device (which is instrumented in terms of hardware) so that an external energy monitoring device[1] is used to measure the energy consumption. Although, this approach produces a precise model for that device [7], the fact is that with the widely adoption of the Android ecosystem makes it impossible to be widely used[2]. Indeed, the model for each specific device has to be manually calibrated!

[1] A widely used devise is available at `http://www.msoon.com/LabEquipment/PowerMonitor`.

[2] In fact, [7] reports the calibration of the power model for three devices, only.

2.2 Power Model: Dynamic Calibration

In order to be able to automatically calibrate the power consumption model of any Android device, we consider a set of training programs that exercises all components of the power model. The training programs also change (over its full range) the state of each component, while keeping the other constant. In this way, we can measure the energy consumption by that particular component in that state. To measure the energy consumption, instead of using an external monitoring device as discussed before, we consider the battery consumed while running the training applications. The Android API provides access to the battery capacity of the device, and to the (percentage) level of the battery of the devices. By monitoring the battery level before and after executing a training application, we can compute the energy consumed by that application. After collecting power traces for all hardware components, a multi-variable regression approach is used to minimize the sum of squared errors for the power coefficient. Fig. 1 shows the architecture of the dynamic calibration of the power model.

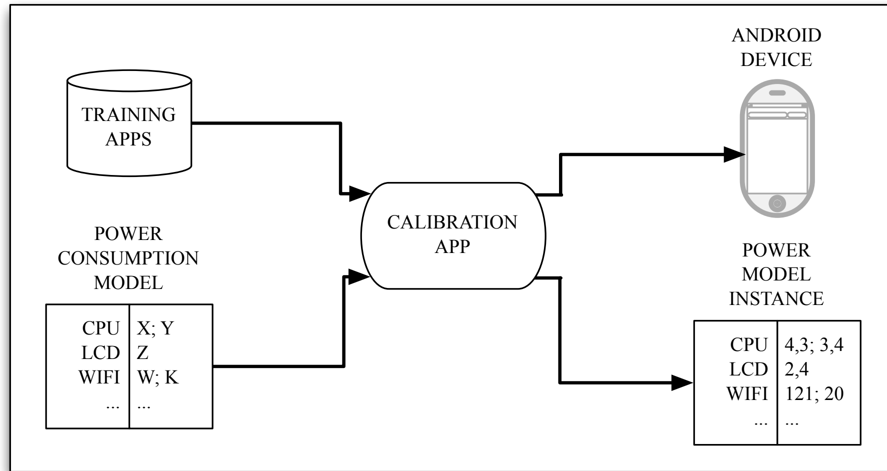

Fig. 1: The architecture to dynamically calibrate the power model for different devices

The calibration process executes the set of calibration applications in a specific device. The generic power model presented in the previous section is then instantiated. The algorithm to dynamically produce such specific model is presented next.

```
public Map<String, Float> calibrate(){
  List<Float> consumptions;
  Map<String, Float> coefficients;
  int N = 20;
```

```
  float after, before;
  float capacity = getBatteryCapacity();
  for(Training app : allTrainings){
    for(State st : app.getStates()){
      consumptions.clear();
      for(int i = 0; i<N; i++){
        before = checkBatteryStatus();
        st.execute();
        after = checkBatteryStatus();
        consumptions.add((after-before)*capacity);
      }
      coefficients.put(st.getName(), mean(consumptions, N));
    }
  }
  return coefficients;
}
```

The result of this algorithm is a collection of energy consumption coefficients, one per state of every hardware component. These coefficients are used to compute the energy consumption of an Android application. For example, when the CPU component is in a known state (i.e., running at a certain frequency, with a known percentage of use), then the power model computes the current energy consumption as a equation of those coefficients. Those Android energy consumption models are often implemented as stand alone applications[3], which indicate the (current) energy consumption of other application running in the same device. In the next section, we present our methodology to use our models in an energy profiling tool for Android application developers.

3 Energy Consumption in Source Code

Modern programming languages offer their users powerful compilers, that included advanced optimizations, to develop efficient and fast programs. Such languages also offer advanced supporting tools, like debuggers, execution and memory profilers, so that programmers can easily detect and correct anomalies in the source code of their applications. In this section, we present one methodology that uses/adapts the (dynamic) power model defined in the previous section, to be the building block of an energy profiling tool for Android applications. The idea is to offer Android application developers an energy profiling mechanism, very much like the one offered by traditional program profilers [12]. That is to say that we wish to provide a methodology, and respective tool support, that automatically locates in the source code of the application being developed the code fragments responsible for an abnormal energy consumption.

[3] Powertutor application website: `https://powertutor.org`.

Our methodology consists of the following steps: First, the source code of the application being monitored is instrumented with calls to the calibrated power model. Fig. 2 displays this step.

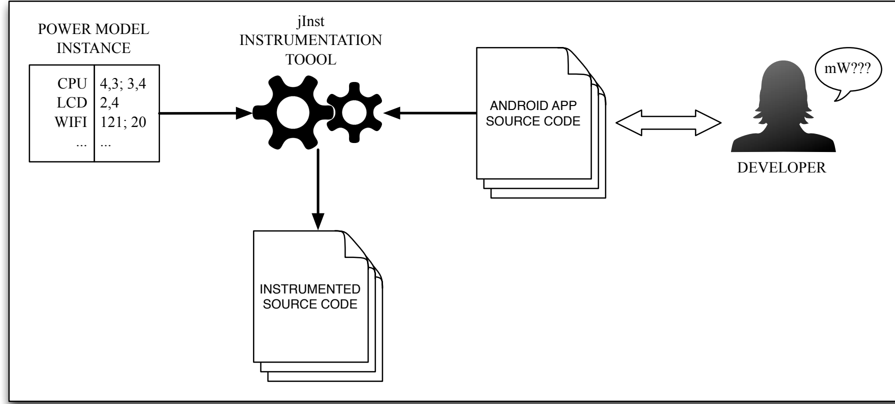

Fig. 2: The behavior of the instrumentation tool

After compiling such instrumented version of the source code, the resulting application is executed with a set of test cases. The result of such executions are statistically analyzed in order to determine which packages/methods are responsible for abnormal energy consumptions.

The source code instrumentation and execution of test cases is performed automatically as we describe in the next Sections. To instrument the source code with calls to the power model, we need to model it as an API. This is discussed first.

3.1 The Model as an API

In order to be able to instrument the source code of an application, with energy profiling mechanisms, we need to adapt the current implementation of power model described in section 2.1. That power model is implemented as a stand alone tool able to monitor executing applications. Thus, we needed to transform that implementation into an API-based software, so that its methods can be reused/called in the instrumented source code.

To adapt the power tutor implementation, we introduced a new Java class that implements the methods to be used/called by other applications and respective test cases. Those methods work as a link interface between the power consumption model and the applications source code to be monitored.

The methods implemented in the new `Java` class, called *Estimator*, and that are accessible to other applications are:

– traceMethod(): The implementation of the program trace .
– saveResults(): store the energy profile results in a file.
– start(): start of the energy monitoring thread.
– stop(): stop of the energy monitoring thread.

3.2 Source Code Instrumentation

Having updated the implementation of the power model so that its energy profiling methods can be called from other applications, we can now instrument an application course code to call them.

In order to automatically instrument the source code, we need to define the code fragments to monitor. Because we wish to do it automatically, that is by a software tool, we need to precisely define which fragments will be considered. If we consider code fragments too small, for example, a line in the source code, than, the precision of the power model may be drastically affected: a neglected amount of energy would probably be consumed. In fact, there is not a tool that we can use capable of giving power consumption estimates at a so fine grained level, with reliable results. On the other hand, we should not consider to large fragments, since this will not give a precise indication on the source code where an abnormal energy consumption exists.

We choose to monitor application methods, since they are the logical code unit used by programmers to structure the functionality of their applications. To automatize the instrumentation of the source code of an application we use an open source JavaParser tool[4]: it provides a simple Java front-end with tool support for parsing and abstract syntax tree construction, traversal and transformation.

We developed a simple instrumentation tool, called jInst, that instruments all methods of all Java classes of a chosen Android application project, together with the classes of an Android test project. jInst injects new code instructions, at the beginning of the method and just before a return instruction (or as the last instruction in methods with no return), as shown in the next code fragment:

```
public class Draw{
...
public void functionA(){
  Estimator.traceMethod("functionA", "Draw", Estimator.BEGIN)
    ;
  ...
  Estimator.traceMethod("functionA", "Draw", Estimator.END);
}
```

This code injection allows the final framework to monitor the application, keeping trace of the methods invoked and energy consumed.

[4] Java parser framework webpage: https://code.google.com/p/javaparser.

3.3 Automatic Execution of the Instrumented Application

After compiling the instrumented source code an Android application is produced. When executing such application energy consumption metrics are produced. In order to automatically execute this application with different inputs, we use Android testing framework[5] that is based on jUnit.

In order to use the instrumented application and the developed *Estimator* energy class, the application needs to call methods start and stop before/after every the test case is executed. Both jUnit and Android testing framework allow test developers to write a *setUp()* and a *tearDown()* methods, that are executed after a test starts and after a test ends, respectively. So, our jInst tool only needs to instrument those methods so we can measure the consumption for each test, as shown in the next example:

```
public class TestA{
...
@Override
public void setUp(){
  Estimator.start(uid);
  ...
}
...
@Override
public void tearDown(){
  Estimator.stop();
  ...
}
```

With this approach, we assure that every time a test starts, the method *Estimator.start(int uid)* is called. This method starts a thread that is going to collect information from the operating system and then apply the power consumption model to estimate the energy consumed. The *uid* argument of the method is the UID of the application in test, needed to collect the right information. The *tearDown()* is responsible for stopping the thread and saving the results.

3.4 Green-aware Classification of Source Code Methods

Now, we need to define a metric to classify the methods according to the influence they have in the energy consumption. They are characterized as follows:

- *Green Methods:* These are the methods that have no interference in the anomalous energy consumptions. They are never invoked when the application consumes more energy than the average.

[5] Android testing web page: `https://developer.android.com/tools/testing/index.html`.

– *Red Methods:* Every time they are invoked, the application has anomalous energy consumption. They can be invoked when the application has bellow the average energy consumption as well, but no more than 30% of the times. They are supposed to be the methods with bigger influence in the anomalous energy consumption.
– *Yellow Methods:* The methods that are invoked in other situations: mostly invoked when the application power consumption is bellow the average.

This representation of methods is also extensible for classes, packages and projects. In other words, the classification of classes depends on the set of methods they have, and so packages depend on their respective classes, as the projects are classified according to their packages. If a class has more than 50% of its methods classified as Red Methods, then it is a red class. With more 50% of them as green, it is a Green Class. Otherwise, it is a Yellow class. Packages and projects follow the same approach.

4 GreenDroid: An Android Framework for Energy Profiling

At this point, we have power source code instrumentation, consumption measuring, method tracing and automatic test execution using the Android testing framework. The final result should be a tool that works automatically, and does all this tasks incrementally. After that, it retrieves the information previously saved and shows the results obtained. This section explains the workflow of the framework, along with the information obtained at every point, and how the results are obtained and generated.

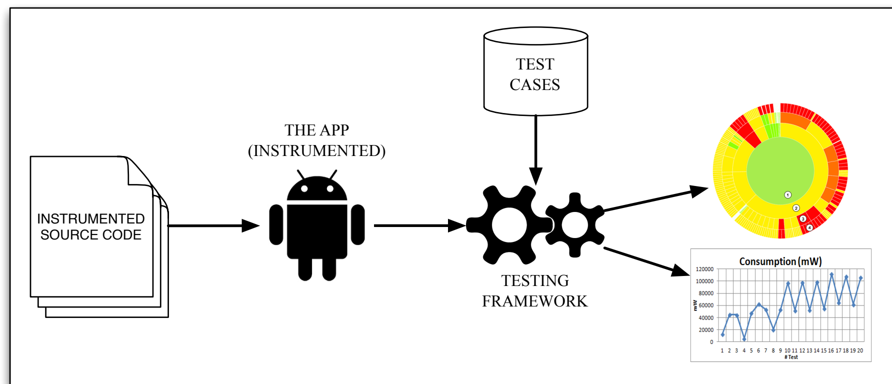

Fig. 3: The behavior of the monitoring framework

4.1 Workflow

After the source code of the application and the tests are instrumented as described in section 2, the framework executes a set of sequential steps to show the conclusive results, that being:

- *Execute the tests*: This is the starting point for the framework. The tests will be executed twice, the first time to get the trace (list of invoked methods) and the second to measure power consumption, so that the tracing overhead does not affect measuring. The results will be saved in files (one for each test), containing a list of the methods invoked, along with the number of times it was invoked, and also the execution time of the test and the energy consumed, in mW.
- *Merge the results*: After all the tests executed (twice), the framework would have generated a set of files as big as the number of tests. For convenience, the tests will be merged in one file to be read, parsed and the information extracted once.
- *Classify the methods*: At this point, the framework will get the values read from the file and classify them (and respective classes, packages and projects) according to the categories described in section 3.4.
- *Generate the results*: The framework will than generate a graphical representation of the source code components, giving them different colors according to its green-aware classification.

This steps are all represented in Figure 3, that represents how the application, after instrumented, generates the results for measuring and tracing of the test cases defined with the Android testing framework.

5 Results

This section shows the results of running multiples tests with our framework from an open source Android application called *0xBenchmark*[6]. This application allowed us to simulate different kinds of executions. We managed to run 20 different tests, and each test had a different set of methods invoked (execution trace). So, for each test we managed to keep the trace, the power consumption, the time a test executed and the number of times a method was invoked. It is important to refer that the values presented in the charts reflect, for each test case, an average of several measures. It makes sense to do it since this is a statistical approach. If we look at the Fig. 4, we can see that different tests have different values of power consumption. One could think that the tests with bigger values for power consumption are the ones with bigger execution times, and so the energy consumed per unit of time would be nearly the same for all the tests, but Fig. 5 shows that the consumption per second varies between the tests.

[6] 0xbench can be found at `http://0xbenchmark.appspot.com`.

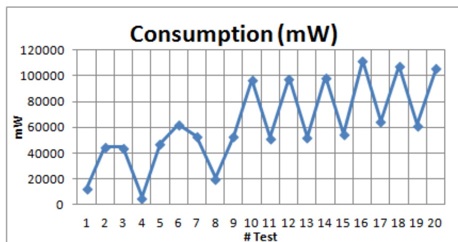

Fig. 4: Total consumption per test

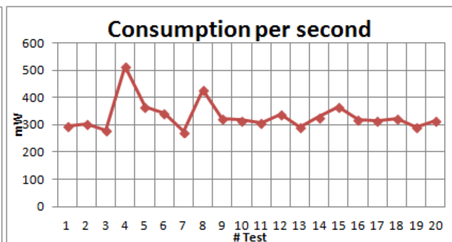

Fig. 5: Consumption per second

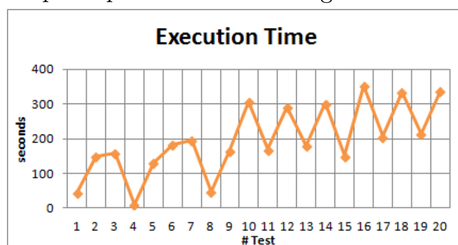

Fig. 6: Execution time

So this are very good indicators, they allow us to conclude that execution time has an influence in the power consumption, but it is not the only relevant factor. In fact, it might not be one of the most relevant.

So, with the approach described in section 3 to detect tests with excessive power consumption, we can get a percentage for each method that reflects it's influence in energy anomalous tests. The framework will then assign that percentage to the respective method, taking into account the test results, and display a sunburst diagram like the one in Fig. 7 (similar to the approach presented in MZoltar [13]) that allows the developer to quickly identify the most energy anomalous methods.

6 Related Work

Power consumption analysis of Android applications is not an unexplored area in the investigation scope. In the past years, the investigation in the *smartphones* power consumption has been increasing. There are a lot of research works indicating that power consumption modeling and energy-aware software are getting their importance in the investigation scope. We can find different tools designed to estimate the required energy for an application to do it's tasks. The majority of them focus on the Android based *smartphones*, mostly because it's an open source OS[7] and statistics reveal that the percentage of selling is much bigger for

[7] An Android overview can be found at `http://www.openhandsetalliance.com/android_overview.html`.

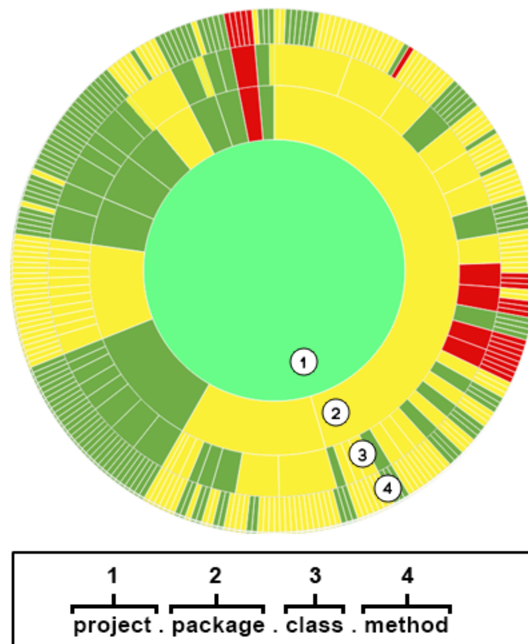

Fig. 7: Sunburst diagram (and how to interpret it)

Android devices than iOS devices (iPhone)[8]. In fact, in the second quarter of 2013 almost 80% of the market share belonged to Android devices.

Most of the research works starts by identifying the hardware components of the device with significant influence on it's energy consumption. We have the example of Power Tutor [7], that is the starting point for many other research works, but also DevScope [11] and related tools (AppScope [10] and UserScope [14]). This tools have a power consumption model relating the different hardware components of a device to it's different states and consequent power consumption values. The main difference lies in the implementation of the model: one works as an independent Android application, the other is a linux kernel module, but both of them focus on collecting the hardware components' usage information from the operative system.

There are other examples of works based on power consumption models and it's applications in different areas ([9, 15–17]), however none of them is as powerful or adjusted to this project necessities/goals as the remaining ones. Another interesting example, SEMO [18], has a similar behavior to Power Tutor, but doesn't use power consumption models. Instead, is focused in battery discharge level, and it's results are a little more unreliable.

[8] Information about global smartphone shipments can be found at `http://techcrunch.com/2013/08/07/android-nears-80-market-share-in-global-smartphone-shipments-as-ios-and-blackberry-share-slides-per-idc`.

Other works [19, 20] demonstrate that is possible to have different values on energy consumption for different softwares designed to do the same tasks, so this can be a very good indicator that helping developers choose the most energy-aware solution for a software implementation is a great contribution.

7 Conclusions and Future Work

The energy consumption is nowadays of paramount importance. This is also valid to software, and specially for applications running in mobile devices. Indeed the most used platform is clearly the Android and thus we have devote our attention to its applications.

Given the innumerable quantity of Android versions and devices, our approach is to create a dynamic model that can be used in any device and any Android system, and that can give information to the developers about the methods he/she is writing that consume the most energy. We have created a tool that can automatically calibrate the model for every phone, and another to automatically annotate any application source code so the programmer can have energy consumption measures with almost no effort.

With this work we were able to show that the execution time is highly correlated to the total energy consumption of an application. Although this seems obvious, until now it was only speculative. We have also shown that the total time and energy the application takes to execute a set of tasks does not indicate the worst methods. To find them, it is necessary to apply the techniques we now propose, measuring this consumption by second and computing the worst methods, called red methods.

Nevertheless, there is still work to be done. Indeed it is still necessary to evaluate the precision of the results of our consumption measurements. Since we do not use real measurements from the physical device components, we still need to confirm that the results we can compute are accurate enough.

References

1. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3) (September 1996) 480–503
2. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. **28**(6) (November 2006) 990–1034
3. Krall, A.: Efficient javavm just-in-time compilation. In: International Conference on Parallel Architectures and Compilation Techniques. (1998) 205–212
4. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science **73** (1990) 231–248
5. Saraiva, J., Swierstra, D.: Data Structure Free Compilation. In Stefan Jähnichen, ed.: 8th International Conference on Compiler Construction, CC/ETAPS'99. Volume 1575 of LNCS. (March 1999) 1–16
6. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. PEPM '11, ACM (2011) 131–140

7. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones". Proc. Int. Conf. Hardware/Software Codesign and System Synthesis (October 2010)
8. Dong, M., Zhong, L.: "Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems". MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services (2011)
9. Kjaergaard, M.B., Blunck, H.: "Unsupervised Power Profiling for Mobile Devices". 8th International ICST Conference, Copenhagen, Denmark (December 2011)
10. Yoon, C., Kim, D., Jung, W., Kang, C., Cha, H.: "AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring". USENIX Annual Technical Conference (USENIX ATC'12) (June 2012)
11. Jung, W., Kang, C., Yoon, C., Kim, D., Cha, H.: "DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components". International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12) (October 2012)
12. Runciman, C., Röjemo, N.: Heap Profiling for Space Efficiency. In Launchbury, J., Meijer, E., Sheard, T., eds.: Second International School on Advanced Functional Programming. Volume 1129 of LNCS. (1996) 159–183
13. Machado, P., Campos, J., Abreu, R.: "MZoltar: Automatic Debugging of Android Applications". First international workshop on Software Development Lifecycle for Mobile (DeMobile), co-located with European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2013, Saint Petersburg, Russia (2013)
14. Jung, W., Kim, K., Cha, H.: "UserScope: A Fine-grained Framework for Collecting Energy-related Smartphone User Contexts". IEEE International Conference on Parallel and Distributed Systems(ICPADS 2013) (December 2013)
15. Kim, D., Jung, W., Cha, H.: "Runtime Power Estimation of Mobile AMOLED Displays". 2013 Design, Automation & Test Europe (DATE'13) (March 2013)
16. Carroll, A., Heiser, G.: "An Analysis of Power Consumption in a Smartphone". USENIXATC'10 Proceedings of the 2010 USENIX conference on USENIX annual technical conference (2010)
17. Zhang, L., Gordon, M.S., Dick, R.P., Mao, Z.M., Dinda, P., Yang, L.: "ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications". IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (2012)
18. Ding, F., Xia, F., Zhang, W., Zhao, X., Ma, C.: "Monitoring Energy Consumption of Smartphones". Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing (October 2012)
19. Corral, L., Georgiev, A.B., Sillitti, A., Succi, G.: "Method Reallocation to Reduce Energy Consumption: An implementation in Android OS". Symposium On Applied Computing '14 (2014)
20. Noureddine, A., Rouvoy, R., Seinturier, L.: "Unit Testing of Energy Consumption of Software Libraries". Symposium On Applied Computing '14 (2014)