Detecting Energy Leaks in Android App with POEM

Alan Ferrari, Dario Gallucci, Daniele Puccinelli, and Silvia Giordano

Networking Laboratory
University of Applied Sciences of Southern Switzerland (SUPSI)
Manno, Switzerland
name.surname@supsi.ch

Abstract—**This paper presents the design and implementation of a Portable Open Source Energy Monitor (POEM) to enable developers to automatically test and measure the energy consumption of every single application component down to the control flow level. Based on existing portable power meter designs, POEM extends the state of the art of application analysxis with the energy annotation of the control flow down to the basic blocks, the call graph, and the Android API calls, allowing developers to locate energy leaks in their applications with high accuracy. Because the power consumption is tied to the system status, energy annotation is also coupled with system activities.**

I. Introduction

In recent years smartphones have become hugely popular, allowing users to access a rich set of functionalities in any condition and location. Despite their performance improvement in terms of computational and storage capabilities, their reliance on batteries remains problematic. Many complaints regarding the battery drain have appeared in the marketplace reviews. This leads the developers to face the challenge to build energy-aware applications. Many solutions to reduce the energy consumption of applications have been proposed in recent years; the majority of them offload part of the computations to the cloud [1] or to neighboring devices [2]. In spite of the advantages of offloading techniques, many energy leaks might be easily solved by improving poorly written applications.

There is a clear need for a detailed understanding of the power consumption of smartphone apps. Due to the poor quality of smartphone internal battery monitoring unit, a set of freely available solutions that help developers in their work have been proposed in the scientific community. Many of them rely on smartphone hardware modelling to retrieve accurate power measurements [3], whereas others rely on external power monitors to do the same job [4]. On the software side there are several solutions to study the application behavior, ranging from static code analysis to the analysis of the system calls [5], [6]. This paper presents the design and implementation of a Portable Open Source Energy Monitor (POEM), a novel solution to overcome the limitations of current approaches. Based on a portable powermeter, whose design is in turn based on Battor [7] and NEAT [8], we extend the state of the art of application analysis with the energy annotation of the control flow down to the basic blocks, the call graph, and the Android API calls, allowing developers to find where the energy is spent inside their app with the maximum possible accuracy. Because the power consumption is tied to the system status (for instance, whether the screen is lit or whether the radios are on), we also synchronize our energy annotation with system activities. Finally, we offer a powerful and dynamic visualization that enables developer to easily spot energy leaks in their apps.

The rest of the paper is organized as follows: The first section provides more insights in the related work and describes the starting point on which we build POEM. The second section illustrates the details of the design and implementation of POEM. The third section describes the evaluation of POEM based on several case studies where we analyzed the power consumption of a set of benchmark problems.

II. Related work

We first focus on the retrieval of accurate power measurements in modern smartphones and continue with an overview of the state of the art of the power consumption analysis of smartphone apps.

A. *Energy measurements on smartphones*

Power consumption in modern smartphones is generally measured with hardware-based or model-based techniques. Hardware-based methods are based on the internal battery unit on the smartphone or external measurement platforms such as the Monsoon power meter [9]. It is notorious that the internal battery unit on mobile devices is inaccurate and in the majority of cases work in a very high sampling rate. On the other hand, external power monitors offer high accuracy and low level sampling rates, but they require tethering the smartphone to a desk, thus keeping the user outside the testing loop and making it impossible to record usage patterns and environment conditions. Model-based [10], [11] techniques partly rely on the internal battery unit (BMU) in the smartphone, but refine the measurements by accurately modeling the internal hardware consumption. Though these techniques offer an acceptable level of accuracy, the complexity of modern smartphones makes it virtually impossible to model every single state [12]. Recently, solutions based on portable hardware have been proposed. The systems described in [7] and [8] provide external portable power monitoring tools based on small custom board with memory and processing capabilities that are connected between the battery and the phone itself. All current flowing between the battery and the smartphone is routed over a shunt resistor that allows the accurate measurement of the current drawn by using the Ohm's Law. All measurements need to be synchronized with the smartphone in order to detect which activities are responsible for a recorded energy pattern. Battor uses predetermined power consumption patterns and NEAT is physically connected to the phone buzzer, which is dedicated to synchronization (with a consequent minor loss of user experience). Our solution provides a fully open source version of the portable energy monitor based on the Arduino[13] hardware platform and non-intrusive LED2LED-based [14] synchronization.

B. Extracting application behavior

Several solutions have been recently proposed to determine where the energy is spent inside an application. AppScope [5] from Yoon et al. physically modifies the Android kernel by adding a logger that monitors software/hardware interactions and provides an estimation model to catch the energy consumption of the application. Despite its advantages, AppScope does not provide any information at the application level (e.g., which method is taking most of the energy). eCalc [15] and vLens [15] focus on the application side and use static code analysis to extract meaningful information about the software as well as code-injection to extend and annotate programs in order to collect information at runtime. The approach in Lit et. al. [6] uses code injection to annotate and extract information from the application and employs linear regression to find the energy consumption at the bytecode instruction level. These methods all provide high accuracy on the application side but fail to consider the control flow of the application (e.g. loops and jumps) and in many cases they have a significant level of inaccuracy.

Our approach focuses on the static analysis of the bytecode and code injection techniques to obtain measurements with several levels of granularity: class level, method level, internal control flow level, and Android API level. On the control flow level we look at the basic blocks [16] and employ the technique known as dominator analysis to extract loops and jumps [17]. To account for the system behavior, we periodically log most system activities (i.e. CPU frequency, network usage, ...) to detect battery leaks (i.e. a basic block overusing network connectivity).

III. System Architecture

There are several levels that are simultaneously approached in our solution in order to provide a useful toolchain for the detection of battery leaks in Android applications:

- At the *hardware level*, we provide a portable external power meter.
- At the *system level*, we provide an Android system logger.
- At the *software level*, we provide an offline code analysis tool to extract the internal properties of the application as well as a code-injection mechanism to annotate the internal state of the application to track what happens at runtime.

Figure 1 shows the path that a developer has to follow in order to use the toolchain. The first step is to use our tool to annotate and extract the meaningful features from the application's bytecode. The second step is to run the application and simultaneously collect the traces with an external power meter and with the phone logger (basic blocks, method usage, and system status). The final step is the analysis and the visualization of the results.

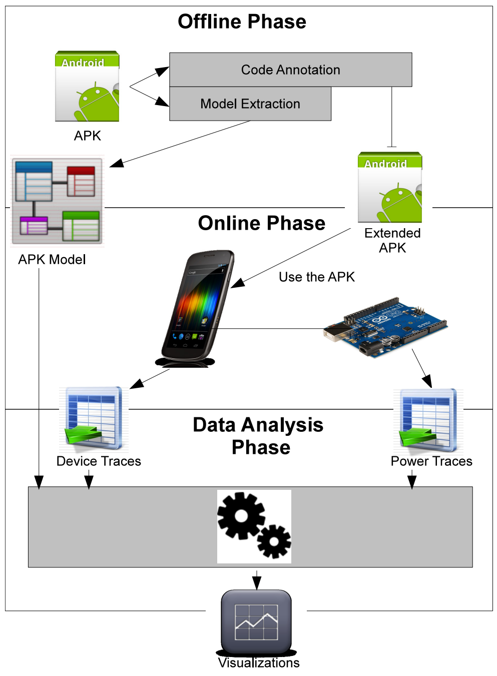


Fig. 1: The various phases needed to analyze the power consumption of a specific application.

A. Hardware Layer

The core of our portable power meter prototype is the Arduino Leonardo board. It uses the ATmega32u4 micro controller with 32KB flash memory and 2.5KB of SRAM Memory. Arduino offers 20 digital I/O pins and 5 Analog pins. The analog pins can read signals from 0 to 5 V and their ADCs have a 10 bit resolution (roughly 5 mV). The minimum reading time for an analog pin is 100 microseconds, which corresponds to a maximum reading frequency of 10KHz.

On the smartphone we use a rooted Galaxy Nexus smartphone [18] equipped with a 1.2 GHz ARM Cortex CPU, 1 GB of RAM, running Android 4.3 Jelly Bean. We place a shunt resistor in series with the phone (a shunt resistor is designed to have a very low ohmic value causing a negligible voltage drop that does not affect the smartphone), and we use the Arduino ADC to read the voltage across the shunt resistor, whose value is equal to 0.1Ω in our prototype. The current through the shunt is the same as the current drawn by the smartphone and can be computed with Ohm's law based on the voltage readings. The corresponding power draw of the smartphone can be obtained as the product of the computed current and the input voltage (3.7 V for our phone).

To improve the granularity of the readings, we use an op amp to amplify the voltage across the shunt (by a factor of $\approx$ 10 in our implementation), and we use a reference voltage of 3.3 V in the Arduino ADC to achieve a resolution of 12 mW. We have extended the Arduino board to save the results to an external SD-Card and we have created an application to track and record the real-time power draw of a smartphone.

Because our goal is to single out the energy bottlenecks of mobile apps, the recorded measurements must be synchronized with the device so we know the energy footprint of the various components of an app. We employ LED2LED communication [14] between an external LED connected to the Arduino and the phone camera LED to initiate and terminate the recording of energy measurements, which are saved to a dedicated file in the Arduino's SD-Card. The Arduino board always begins measuring prior to the phone due to the duration of the LED pulse, but the lead time of the Arduino board has been found to be fairly deterministic.

B. System Layer

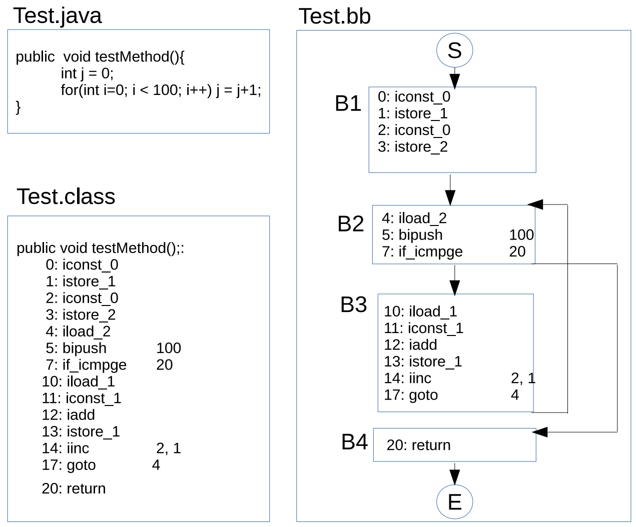
Because many power consumption factors are not directly related to applications (e.g. screen light), we provide a system logger that monitors the phone hardware usage (CPU usage, memory usage, network usage and connectivity, screen light) and stores the system *logcat*, which is used to extract other events (e.g. the calls to the Garbage Collector) and to store the logs of the extended version of the APK (the original APK is extended with code annotations as showed in Section III-C).

C. Application Layer

Following [15], we perform a static analysis of the byte-code to obtain a model that describes the internal properties of the applications (from classes to method control flow). We then use code injection techniques to mark all points of interest inside the application: classes, methods, control flow (basic blocks), and Android API Calls. We pursue the static analysis of the bytecode instead of the standard code annotation for two main reasons. First of all, we may only have access to a compiled version of the app (e.g. downloaded from the marketplace). Secondly, because different compilers may add different optimizations that might change the control flow (e.g. method inlining), adding code lines prior to compile time may affect those optimizations, thus leading to significantly different energy consumption patterns.We automatically extract the call graphs regarding the internal methods of the application. If a method calls an Android API, we store that call and categorize the API according to the approach proposed in [19]. At runtime we collect the number of internal calls that the software is performing and we show the information in the call graph in order to allow the developers to easily navigate through the calls and easily find where the battery consumption is going. For each method we also extract the control flow basic blocks [17], (that may be described as a *a portion of code inside a program that has only a single entry and a single exit point*), A control flow graph may be extracted as shown in Figure 2 and mark each single entry point, which allows us to follow the runtime path in the control flow of the application, thus providing detailed information regarding the power consumption of the runtime path.

Because most of the execution time of an application is accounted for by loops, it is essential to determine whether a certain jump instruction corresponds to a loop. To this end, we employ dominator analysis [20], a well known technique used to retrieve several meaningful pieces of information from a control flow graph.

In a control flow graph, $G = (V, E)$, where V are the basic blocks (nodes) and E are their interconnections (edges),



```
public  void testMethod(){
        int j = 0;
        for(int i=0; i < 100; i++) j = j+1;
}
```

```
public void testMethod();:
    0: iconst_0
    1: istore_1
    2: iconst_0
    3: istore_2
    4: iload_2
    5: bipush          100
    7: if_icmpge       20
   10: iload_1
   11: iconst_1
   12: iadd
   13: istore_1
   14: iinc             2, 1
   17: goto            4
   20: return
```

```
B1
0: iconst_0
1: istore_1
2: iconst_0
3: istore_2

B2
4: iload_2
5: bipush           100
7: if_icmpge        20

B3
10: iload_1
11: iconst_1
12: iadd
13: istore_1
14: iinc            2, 1
17: goto            4

B4
20: return
```

Fig. 2: Example of Basic Blocks Analysis: Test.java shows the standard Java code, Test.class hows the java byte code and Test.bb shows the basic blocks extracted from Test.class

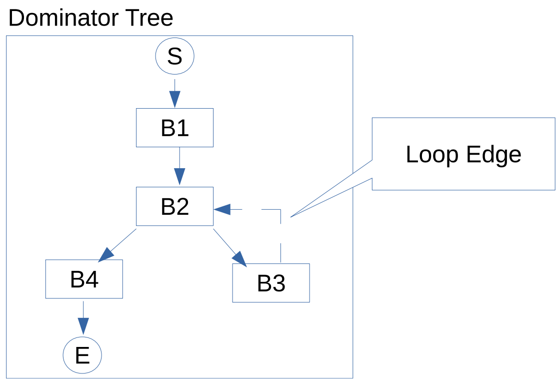


Fig. 3: Example of Dominator Analysis and Loop catching that uses the structures showed in Figure 2 as starting point

a node $d \in V$ dominates a node $n \in V$ if every path from the entry node to n must go through d. Any loop in G has a loop header node that dominates all other nodes in the loop as well as a back edge from a node back to the loop header. Therefore, loops can be identified by identifying back edges in G: first we build a structure known as a dominator tree where there exists a directed edge between nodes $a \in V$ and $b \in V$ if and only if a is the immediate dominator of b (i.e., the closest dominator in G), and then we add all the missing edges that were originally contained in E. A loop exists if any of such edges goes from a node to one of its dominators. An example is shown in Figure 3, where the back edge from B3 to B2 indicates the presence of a loop. Once we know the set of blocks that belong to the loop, we can count the number of iterations for each method call and determine the path followed within the control flow at each iteration. The corresponding energy footprint can then be determined based on the power traces. This approach arguably offers a much more detailed view compared to previous approaches because it accounts for the control flow variations that may occur across different runs due to different usage patterns that translate into different energy consumption profiles.

D. *Visualization*

The Visualization toolkit, provided with the tool, provides and easy and simple way to navigate through classes and explore their details (i.e. control flow on basic blocks or calls per method). Another interface allows users to load and parse the traces collected during real time experiments, furthermore those traces can be visualized in the GUI (in the form of statistics) or they can be exported to be managed with external tools (i.e. Matlab).

IV. Evaluation

It is almost impossible to measure the behavior of a piece of software without affecting its performances. The main goal of this section is to show how much POEM affects the behavior of an Android application

We proceed by using a well known sorting algorithm, namely *Shell Sort*, whose computational complexity is a function of the input size *O(n log n)*. We implement the algorithm in a simple Android application that runs on random generated array with constantly increasing dimension. Increasing the array dimension increases the computational time and thus also the number of records to be extracted by POEM.

Figures 4a-4d show our experimental results. We consider a baseline scenario where the application is run without any modifications and a modified scenario where POEM is employed and use two dimensions of cost: runtime and energy consumption. Figure 4a and 4c show, respectively, the runtime and the energy consumption in the baseline scenario while Figure 4b and 4d show, respectively, the run time and the energy consumption when POEM is used. Runtime and energy consumption are plotted as a function of the input size for an individual method call, which serves as a proxy for complexity. We observe that using POEM results in an increase of both runtime and energy consumption of about four orders of magnitude; this happens because each time a basic block or a method is run POEM needs to access the file system for logging purposes, which results in a deterministic logging cost (in terms of either runtime or energy consumption) C_L. The extra runtime and extra energy consumption represent the cost of gaining major insight into the operation of an application and does not affect comparisons among methods. If we let C_{B_i} be the cost of running basic block B_i (again, in terms of either runtime or energy consumption), then the cost of the baseline scenario is equal to $\sum_{i=1}^{N} C_{B_i}$ and the cost of running POEM on top of the baseline scenario is given by $\sum_{i=1}^{N} C_{B_i} + NC_i$, where N denotes the total number of basic blocks; therefore, the extra cost of running POEM can be subtracted away and does not affect comparisons among methods (in fact, Figures 4a-Figures 4c and Figures 4b-Figures 4d exhibit comparable trends).

V. Case Studies

In order to show the correctness of the behavior of POEM we follow the approach of [15] and build a custom-made application that implements three benchmark problems, namely:

- *B1:* 10x10 matrix multiplication
- *B2:* 10MB file writing on the internal memory
- *B3:* MD5 hashing over a 10 byte string repeated 20 times

To ensure we have a complete understanding of the application and its methods, we use a custom-made application as opposed to a commercial application. The three benchmarks are coded in our custom-made application containing 17 classes (3 of which are the test cases, 1 implements the Main class that calls the test, and the others are Android support classes), and 23 methods. We run each benchmark 100 times.

We start our analysis by considering the APIs called at runtime. As previously mentioned, we classify API calls in 27 categories following [19]. Table I shows the main results; we can immediately discriminate among the three benchmarks saying that the first one is more text-oriented, the second one is IO-oriented and the last one is text- and security-oriented. As shown in Table I, the patterns in the benchmarks are reflected in the API calls: B1 features 11 calls to the *text* API, B2 features 6 calls to the *io* API, and B3 features 2 calls to the *security* API and 2 to the *text* API.

	Global	B1	B2	B3
security	2	0	0	2
appwidget	1	0	0	0
os	2	0	0	0
text	26	11	0	2
io	9	0	6	0
view	2	0	0	0
util	1	1	0	0

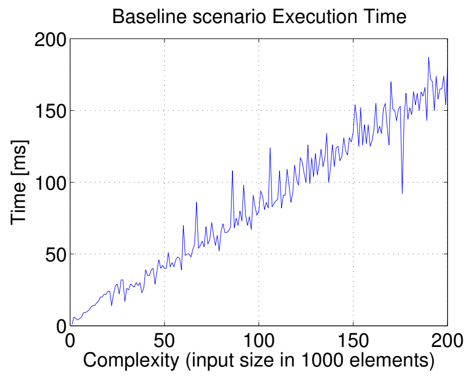
TABLE I: Distribution of API calls in categories (unused categories are: media, nfc, speech, bluetooth, wifi, content, math, dalvik, database, animation, support, telephony, graphics, opengl,net, webkit, test , location, gest, hardware

We continue by extracting the internal method characteristics using basic block analysis. The results are shown in Table II. We immediately see that the most computationally intensive benchmark is B1 with 7 loops, 12 blocks, and 10 branches. We want to point out that the 7 loops are not directly part of the algorithm but some of them are needed to randomly generate two matrices.

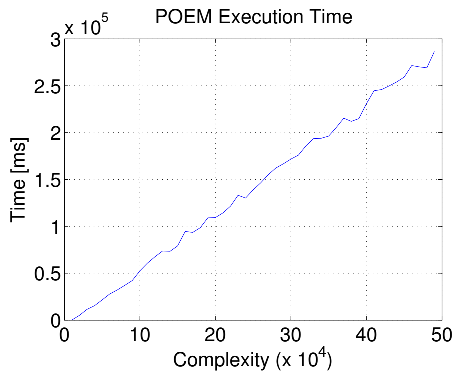
	Global	B1	B2	B3
block	292	12	4	4
branch	21	10	1	1
loop	10	7	2	3
calls	5	0	0	0

TABLE II: Internal method characteristics retrieved with the basic block analysis

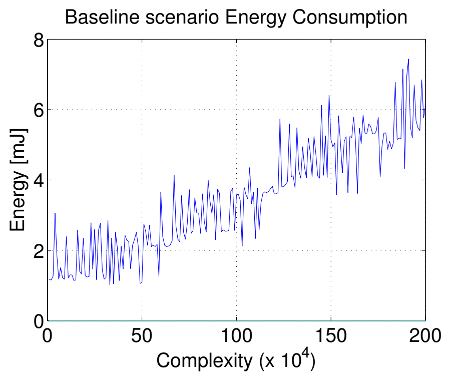
The costs in terms of runtime and energy consumption are shown in Figure 5. We observe that B2 consumes more time and energy compared to others, which can be explained by looking at its I/O operation. Though B2 has only 2 loops, it accesses the I/O as many as 6 times. We also observe that



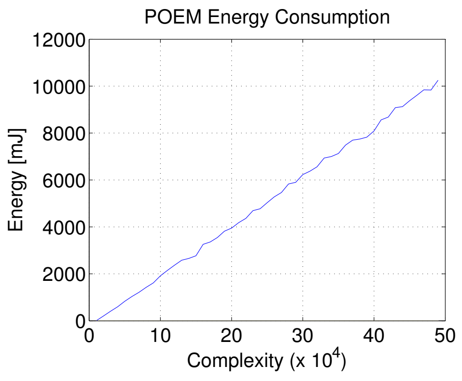
(a) Execution time on the clean application



(b) Execution time on the modified scenario (POEM)



(c) Energy consumption on the clean application



(d) Energy consumption on the modified scenario (POEM

Fig. 4: Experimental results on Shell Sort example.

B1 requires more energy than B3 due to its extra CPU usage. We also observe that that all confidence intervals are relatively narrow for B1 and B3 and wider for B2, which suggests extra system activity triggered by B2's buffered writes.

Our analysis on three benchmarks is meant to give an idea of how POEM can be used to learn where exactly energy gets used in an application (energy leaks). After finding energy leaks, a developer may take appropriate action.

A. *Impact of system states*

It is very difficult to compare different software entities due to the many variables at play. For instance, different hardware states result in different energy footprints (i.e. reducing the screen light dramatically decreases the power consumption). As noted earlier, our solution consists in logging the system status (i.e. screen lighting) in order to differentiate among different methods or block calls. A problem that is not addressed with standard logging techniques is the presence of energy tails due to unclean state transitions, which depend on the internal characteristics of smartphones. The presence of such tails may affect the estimation of the energy usage of whatever task happens to overlap with energy tails. Currently, we address this problem by disallowing comparisons in two cases: If one of the blocks to be compared is run immediately after any hardware component has changed status, and if the one of the blocks to be compared is run in a different hardware state than the other block (i.e. with different screen lighting). Residual errors may persist because we cannot expect to capture every possible event that may take place. We therefore suggest multiple test runs to isolate out outliers and ensure consistency.

VI. Conclusion and Future Work

This work presents POEM, a portable and open source energy monitor that allows developers to automatically test and measure the power consumption of every single application component down to the control flow level. We use basic block analysis to determine the control flow (branch and loops) of every single method in order to find computationally intensive tasks and discriminate among other activities (i.e. I/O tasks). We are currently extending and improving POEM to meet the needs of developers. Therefore, we are releasing POEM as a open source project and welcome community contributions. One of our next step is to understand whether the energy footprint of a method can be assessed solely based on its control flow and complexity estimation (i.e. with Cyclomatic complexity [21]), thus bypassing the need for physical energy measurements though our initial POEM prototype is already portable, its logging is very power-intensive and therefore makes our prototype challenging to use in real-world deployments. We are considering the use of techniques such as Bursty Tracing [22], [23] that reduce the energy cost of logging by selectively calling the logger only at a fixed intervals and would greatly simplify the adoption of POEM in real-world deployments.

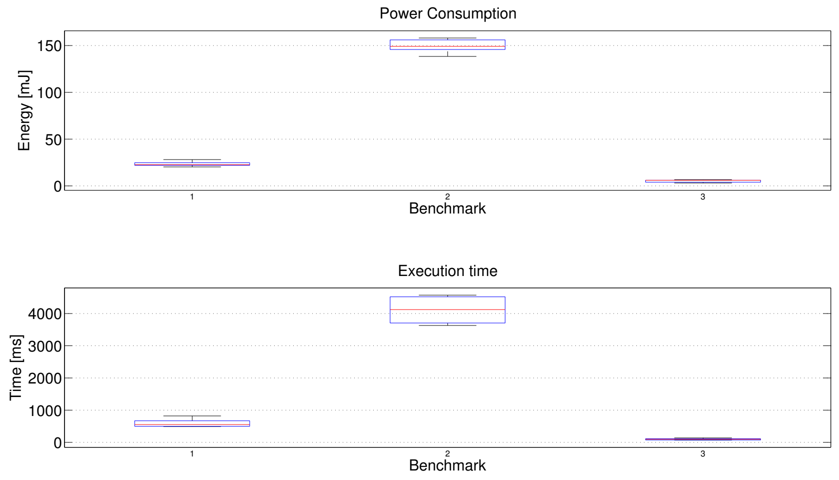


Fig. 5: Costs in terms of runtime and energy consumption observed during the runs of our three benchmarks

Acknowledgements

This work was supported by the EU FP7 ERANET program under grant CHIST-ERA-2012 MACACO.

References

[1] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *MobiSys'10*, San Francisco, CA, USA, June 2010.

[2] A. Ferrari, D. Puccinelli, and S. Giordano. Code Offloading on Opportunistic Computing (demo paper). In *IEEE International Conference on Pervasive Computing and Communications (PerCom'14)*, Budapest, Hungary, March 2014.

[3] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM, 2011.

[4] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[5] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, 2012.

[6] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.

[7] Aaron Schulman, Thomas Schmid, Prabal Dutta, and Neil Spring. Demo: Phone power monitoring with battor. In *MobiCom (Conference on Mobile Computing and Networking)*, 2011.

[8] Marco Zuniga Niels Brouwers. Neat: A novel energy analysis toolkit for free-roaming smartphones. In *Under Submission*, 2014.

[9] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor, May 2014.

[10] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348. ACM, 2011.

[11] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 353–362. ACM, 2012.

[12] John C McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C Snoeren, and Rajesh K Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, 2011.

[13] Arduino: Open-source Electronics Prototyping Platform. http://www.arduino.cc, May 2014.

[14] Domenico Giustiniano, Nils Ole Tippenhauer, and Stefan Mangold. Low-complexity visible light networking with led-to-led communication. In *Wireless Days (WD), 2012 IFIP*, pages 1–8. IEEE, 2012.

[15] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 1–7. IEEE, 2012.

[16] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[17] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[18] Google Nexus Smartphones and Tables. http://www.google.com/nexus, October 2013.

[19] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[20] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[21] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, 1(4):308–320, 1976.

[22] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001.

[23] Matthias Hauswirth and Trishul M Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ACM SIGPLAN Notices*, volume 39, pages 156–164. ACM, 2004.