Navitas Framework: A Novel Tool for Android Applications Energy Profiling

Vladislav Myasnikov*, Alexey Shaposhnikov† and Stanislav Sartasov‡,
Egor Gordienko§, Olga Aphonina¶ and Alan Gamaonov‖,
Saint Petersburg State University
* vladislav.myasnikov@bk.ru
† st069259@student.spbu.ru
‡ stanislav.sartasov@spbu.ru
§ egor.gordienko.00@gmail.com
¶ o.aphonina@gmail.com
‖ st061582@student.spbu.ru

***Abstract*—In a modern world smartphones became a commonly used electronic devices performing numerous day-to-day tasks and much more. But they require battery power to operate. It is well-known that computationally intensive programs as well as those using different smartphone peripherals tend to discharge the battery much quicker then their less intensive counterparts, leading to a decreased operating time. To make an application energy-aware, developers need tools to analyze its energy consumption. In this paper we present an open-source software framework to create such tools, *Navitas Framework*, as well as its practical application — an Android Studio IDE plugin to profile energy consumption of an application, *Navitas Profiler*. We describe design and architecture of the framework, outline plugin capabilities and demonstrate its usage.**

***Keywords*—energy efficiency, green software engineering, mobile development, energy profiling, Android, software power metering**

I. Introduction

"My phone discharged!" In a modern world this phrase is a known source of frustration for billions of people. By the end of 2023 it is projected for a number of smartphone users to reach 4.3 billion with Android OS still being a leading mobile OS [1].

As smartphone components require electrical power to operate, a battery provides a fixed level of voltage and a variable current. When battery charge level is low, voltage level drops beyond a certain threshold, where it is not enough for smartphone components to work properly. As smartphone peripherals tend to consume more power during high workloads than during idle states, it can be seen why computationally intensive or network-intensive software spend battery charge more rapidly than its less intensive counterparts. The less time remains for the user to work with an application, the worse the user experience.

While advances in materials and electronics in the last years helped to offset this problem by introducing more capacious batteries or power-saving processors, the issue of energetically inefficient software is still important. Green software development views energy consumption considerations as important as performance metrics [2], but it is still not so popular amongst developers [3]. One of the valuable results in this field are energy-efficient refactorings — code changes that don't change application behavior, but reduce its energy footprint [4].

But how bad is particular code from an energy consumption perspective? Do we really need to apply a refactoring to it? To answer these questions a developer needs to measure power drain of an application, module, procedure or test, therefore a power metering tool which is reliable and easy to use is required. We think that such a tool should become an integral part of development culture instead of being used only during energy bug fixing sessions.

Some successful frameworks and tools were created previously, but we identify two essential gaps we would like to fill with our software:

- Overall level of IDE integration is still regretfully low. Only a handful of tools can be run inside IDE, and among those who can, not every profiling scenario useful to developers is supported.
- To our knowledge, no power metering software considers multi-threading to be an important factor. Multi-threaded applications, while being common under mobile operating systems for more than a decade, introduce an additional level of complexity to energy consumption measurement or estimation.

With this considerations in mind we present *Navitas Framework* and *Navitas Profiler* plugin for Android Studio IDE.

This paper is organized as follows. In Section 2 we lay out the context of our work in terms of Android OS capabilities along with the related works. In section 3 the design of *Navitas Framework* is discussed. Section 4 adds more details to specific modules of a framework and describes the capabilities of the *Navitas Profiler* plugin. Experiments used to prove framework validity are described in Section 5. In Section 6 we list current limitations of our framework and outline future work.

II. Overview

A. Related works

We conducted a systematic literature review (SLR) on the power metering frameworks for Android OS [5]. The main classification point for different approaches in software energy consumption estimation is whether power is directly measured or indirectly estimated.

Direct measurement means that some metering agent, either internal or external to Android device, is directly measuring voltage, current or even power. The simplest way is to connect external digital multimeter to battery contacts of a smartphone and work with an application in question or launch a unit test. At the same time multimeter will write power readings [6], [7], [8]. Alternatively one can use internal power sensors of a smartphone itself [9], [10], [11]. Such a tool can get a good power consumption reading for entire smartphone, experimental design may narrow it down to a level of a single application. For a more detailed report the source code is instrumented with additional logging instructions to mark method or code beginning and end, and two sources of data — power readings and application execution trace [9], [4] — are generated and aligned. Instrumentation energy overhead should also be estimated and accounted [4].

Indirect estimation means that profiling software is aggregating some code execution information and relates it to energy consumption using some model, therefore we also call this approach model-based.

One of the ways to build a model is to estimate bytecode energy consumption by evaluating an energy consumed by a number of specific instruction types using weighted sum (for example, conditional statements, loops, float-point arithmetic etc.) [12]. Direct measurement framework can be used to obtain weight coefficients. This approach allows to evaluate energy consumption of a test run without even running it. It can be further advanced by accounting library functions [4].

Another way to make an indirect estimation framework is to base an estimating model on time percentage that specific hardware component was active or worked at a specific frequency from the finite set [13]. Each value is multiplied by regression coefficient, and the total sum is found. While it is not difficult to track system events like peripheral switching on and off to aggregate this data, the challenge lies in CPU power estimation. Let's model CPU power consumption as a sum of idle consumption and active per core consumption. For the sake of discussion let's assume those values are constant. CPU consumes more power when multiple cores are active compared to a single-core computation. Due to the Peukert's law [14] the real battery capacity under higher power consumption is smaller than under a lower one, and this difference is considerable [13]. Thus using a single coefficient for each instruction type is insufficient for accurate power consumption estimation in a multi-threaded case. A model to account this effect requires power drain value for each combination of peripherals currently turned on [13]. This information is challenging to be obtained experimentally, requires direct measurement framework and significantly complicates model computation.

B. Industrial grade frameworks

Only a handful of the reviewed frameworks has a source code or an application available for reuse, and even less support modern versions of Android OS. However, there is a number of tools that are used in the industry to assess application energy consumption.

BatteryHistorian [15] is a background system events analyzing utility for Android OS. It is a direct measurement framework using internal smartphone sensors through OS API, but energy consumption is represented as a battery charge percentage which is prone to battery degradation and not precise enough to estimate power drain for a short-running test.

Android Studio Energy Profiler [16] is a plugin for Android Studio IDE. It shows power drain in relative terms during application execution. While the information on the model itself is very sparse, we consider it to be indirect estimation framework based on a component working time model. We carefully note that this model is device-agnostic and it is not clear how accurate does it represent real device energy consumption.

Our conclusion is that while there were interesting research projects, and some of the tools see practical use, only Android Studio Energy Profiler is integrated with Android Studio IDE, but it has its own limitations.

C. Multi-threading in Android OS and its challenges to energy consumption measurement

Android OS fully supports multi-threading. Multi-threaded applications are common nowadays, as it is a good way to distribute computational load among the processor cores. This idea became even more lucrative with the advent of big.LITTLE2 processor architecture [17], where processor cores are no longer homogeneous, but instead they are grouped into clusters based on their computational capabilities and power requirements.

Every Android application is being executed as a separate process having its own threads. A process currently running in foreground or able to appear on the screen has its threads assigned to foreground control group, while other processes have their threads assigned to background control group. It is important to note a heavy disparity in processor time between those groups: foreground threads get about 95% of time, while background threads get the rest of it.

To our knowledge, none of the energy consumption frameworks specifically target multi-threaded applications, and only an approach from a single paper can be directly extrapolated to multi-threaded case [13], however the corresponding source code or an application is not available. It is a challenging task to understand individual thread impact on overall energy consumption. It is possible to understand when the thread was actively executing by `ftrace` system calls [18] and to

homogenize peripheral energy consuption over time by controlling DVFS governor. However, as noted previously, CPU power consumption greatly varies depending on a number of cores under active payload. It is currently an open question if there is a correlation between this variable power consumption and the data in CPU power profile and, more importantly, how to adjust measurement and analysis processes to account this variation. To our knowledge, no other papers or profiling tools documentation answer these questions as well.

III. *Navitas Framework* DESIGN

A. Conceptual approach

Direct measurement approach can easily be extended to a multi-threaded case. If program execution trace is not used in the framework, then application energy consumption is an aggregate of energy consumption for all its threads, so the measurement process would not be different from single-threaded case. While our understanding of underlying energy consumption would be limited, in some scenarios this is exactly what is required — to measure total power footprint of an application. If execution trace is used then thread ID information can be added to each trace entry, and by splitting execution trace by thread IDs one may obtain traces for each thread in application. Aligning this information with multimeter power readings will give us a timeline of energy consumption with respect to modules, methods or functions being executed at that time.

A key component for the metering software using a direct measurement approach is a hardware sensor. Sensors capturing momentary current and voltage can be used for power sensing as well as dedicated power sensors. In general, smartphone electronics operates at a constant voltage, and a good power estimate could be achieved by using only an ammeter and a manufacturer specification for peripheral working voltage as a constant[1]. If such a device is integrated in a smartphone itself, we call it an internal power sensor, otherwise it is an external sensor.

Android OS provides a number of APIs for getting power reads from internal sensors. With a root access to a smartphone a developer may directly read files from `proc` folder for momentary power state [7]. Alternatively, one can use `batterystats` power logging utility and download it after test case is finished [19]. Both these options allow a high degree of customization, i.e. they may or may not include information of currently active peripherals like Wi-Fi or GPS modules.

External sensors like Monsoon [20] cost money to acquire, so their acquisition might be a limiting factor. Being an external device to a smartphone they don't capture information on active peripherals at the time of metering. When aligning application traces with external measurement data, in order to properly synchronize time marks one should synchronize timings between different devices [7].

[1]Battery voltage level drops with the discharge, but it is pretty constant for Li-Ion and Li-Pol batteries from 90% to 20%

Both types of sensors can produce a series of time marks and instant power readings. The main deciding factor is the frequency of analog-to-digital converters. For external sensors a typical value is 10 KHz, therefore power difference at intervals of 100 μs can be captured. This frequency is considered in the literature as appropriate for capturing code power consumption on a method level [12], and in our observations the most power-consuming methods typically run at least for a 1 ms or in loops lasting longer than that. In the case of internal power sensing frequencies of 1 Hz or less are not unheard of [21], and that severely limits a number of profiling scenarios.

For the model-based approaches, time-based models are already well-suited for multi-threaded scenario, however special calibration should be done to address simultaneous multi-core processor and graphic card load, as their energy consumption might change non-linearly when a number of active cores or GPU multiprocessors increases. As for the models based on instruction or function energy consumption estimation, this approach is limited in multi-threaded environment. Power coefficients for specific instructions may not reflect underlying peripherals usage. In real life scenarios programs are interacting with smartphone peripherals which are at least as consuming as CPU [22] and are independent from processor execution. While a function call to decrease the screen brightness might be estimated in terms of power drain, the side effect of reducing screen power footprint cannot be reduced to processor instruction energy consumption. Second, library functions power consumption is often dependent on its input length (i.e. data transmission functions) and cannot be accurately presented by linear coefficient.

The following points were the most important when we made a decision on the conceptual approach for the *Navitas Framework*:

- External sensors require money to acquire and a certain qualification to be properly used.
- While direct measurement approach using internal sensors was promising at first, we quickly found out that on our test devices sensors' frequency was not stable and high enough for practical use.
- Instruction and function-level power estimation have inherent conceptual flaws for multi-threaded analysis.
- Time-based models often have a support from manufacturers in the form of `power_profile.xml` — an XML file containing the values of electrical current for various modes of operation of the devices installed in the smartphone. This file can be obtained without root access and can be used as an initial set of weighted coefficients for our model.
- Execution data for time-based models can be obtained using Android API calls and utilities.

In the end, we choose *Navitas Framework* to follow time-based indirect estimation approach.

B. Reporting units

From our SLR we knew that there is no uniformity in reporting units that other frameworks use [5]. Some frameworks

used absolute values like Joules, Watts or Amperes, other used battery discharge percentage or relative non-dimensional units [16]. As raw data from `power_profile.xml` contained values measured in milliamperes (mA), voltage can be found in device specification, and execution time is obtained from test or application execution trace, Joules and battery charge percentage were two main reporting units candidates.

Each recharge cycle for Li-ion batteries inevitably shortens their discharge time under the same drain, therefore estimating battery time change can only be done with a context of how old a battery is and how many recharge cycles it was run through [14]. As controlled experiments tremendously differ from real-time smartphone usage where screen brightness is high and multiple peripherals may be active at the same time, high battery drain might be observed. Because of Peukert's law resulting battery time from such a drain can be estimated, but it will be lower than in experimental environment. We conclude that reporting in Joules results in a simpler estimation model.

C. Instrumentation and its overhead

Source code instrumentation is not necessary if a it can be manually updated with all necessary logging information, however automated approach is preferred, for instance by applying source code transformation using `javassist` library [23] or BCEL [24]. We concur with the literature that method start and end should both be instrumented to provide a proper call hierarchy [9]. Instrumentation trace from a code execution should also be profiled for energy consumption in order to deduct it from total power spending for better metering accuracy [4]. For multi-threaded case instrumentation code should be adjusted to include a thread ID for each log entry. We decided to calculate contribution of each device at the measurements results analysis phase.

D. Tests vs. application runs

There are two ways of deploying an instrumented application for power profiling ([25], [7], [6]). Tests are designed for conducting repeatable executions of the same scenario under the same conditions, and they are more suitable for controlled experiments. Different test runners like `MonkeyRunner`, `JUnit` or `Espresso` can be used for Android applications ([7], [12]). Application run means launching the entire instrumented application on a test device and conducting some manual interaction with it. In general it can't be as repeatable as a test due to human involvement, but it helps to make an overview of general application power drain. A the same time tests may be checking application in an uncharacteristic way compared to a real usage.

Overall, in our opinion the choice between application runs and tests lies in a particular profiling scenario, because the boundary between the use cases for both approaches is blurry. While *Navitas Framework* is agnostic to the level of workload execution, we choose test runs due to simplicity of integration.

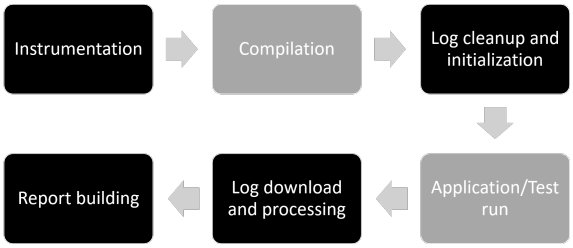

Fig. 1. *Navitas Framework* workflow. Grey background is for Android Studio tasks, black background is for the framework tasks

IV. *Navitas Framework* AND *Navitas Profiler* IMPLEMENTATION

A. High-level overview

Android Studio is currently one of the most popular IDEs among Android developers [26], and while at the time of writing it is a natural choice to develop a profiling tool for it, a more general approach would be to develop a framework based on the current compiling tools and practices and a specific extension or plugin for Android Studio.

Internally, Android Studio runs a set of Gradle tasks to compile and run Android applications. Those tasks are IDE-agnostic and can be reused in other IDEs. We extend this set of tasks with *Navitas* profiling tasks, and the overall workflow can be seen in Fig. 1.

Our first custom task instruments the application code at the compilation stage with additional instructions to log execution trace and thread information. After the compilation is completed, we additionally initialize target device logging facilities to trace component activity information. Then after the test run is finished we download these logs along with execution traces to a computer and format component usage statistics report.

Various test runners can be supported independently from power profiling Gradle tasks. Currently we use `JUnit` as a default test runner, and its capabilities are fully supported. Both single tests and test sets can be profiled, and the profiling results are shown for each test (see Fig. 3). Proof of concepts were also made for `Espresso` and `MonkeyRunner` frameworks.

Navitas Profiler plugin for Android Studio is an orchestrator. It manages what code to instrument, what tests to run and what device to use for it. It also handles the energy coefficients for target devices and it also runs the model against obtained component usage statistics and displays energy consumption report to the user.

B. Framework instrumentation

The instrumentation of application code occurs during compiling and packaging. We used `Transform API` library which allows third-party plugins to manipulate compiled files before they are turned into `dex` files.

To transform the code one needs to create a class and implement `Transform` interface to register the code transformation. The main instrumentation logic is implemented

in the overridden `transform` method. For each of the instrumented files, the necessary imports are added to the beginning. Furthermore, for all methods of each class, methods of the `Javassist` library are applied — `insertBefore` and `insertAfter`. A string containing Java code is passed as a parameter. Even though Kotlin is now the primary development language for Android, many applications are still written in Java, so adding Java code allows us to ensure the correct operation of the original programs in both languages, as full interoperability is maintained between Kotlin and Java. Our experience shows that code instrumentation at compilation phase is faster than APK disassembling and instrumenting mentioned in earlier works.

`Logcat` is a command line tool that prints a log of system messages, including a stack trace, when the device generates an error, and other messages may be written by a custom application using the `Log` class. `Logcat` is convenient to use because in addition to the message itself it adds information about the current time (up to milliseconds), process ID and the thread ID. The code we instrument into methods adds method name to execution trace logs and accesses system API to trace component activity. Activity data is then passed to `Logcat`.

C. Framework component activity data acquisition and model formulas

Among the components of a smartphone which affect energy consumption the most important is the processor. Android OS has special `time-in-state` files in the `/sys` directory that contain information about the time each processor core spent at a specific frequency. The data in these files is stored in pairs like `<frequency><time>`". There are exactly as many of these pairs as there are different frequencies a particular processor core supports. Time is measured in 10 milliseconds units, and it is counted from the moment the corresponding driver was installed to measure processor data.

Our model for the processor energy consumption is therefore straightforward: for each core and for each frequency it operated, one needs to take the time difference between method ending and beginning at the end of the method and at its beginning. Thus we obtain the time method worked at each of the frequencies. Then by multiplying those timings to corresponding weight coefficients in a model and summing the result we obtain the total energy spent by the processor to execute the measured method. Currently, the model weight coefficients are based on the data in `power_profile.xml`. To get the coefficients we multiply the values stored there in milliamperes (mA) to the nominal CPU voltage. In the end the final formula for the CPU power consumption is the following:

$$E_{CPU} = \sum_{i \in cores} \sum_{j \in freqs_i} (endTime_{ij} - startTime_{ij}) * C_{ij} \quad (1)$$

where $startTime_{ij}$ and $endTime_{ij}$ — the time the *i*-th kernel stays at the *j*-th frequency at the time of entering and exiting the method, respectively, C_{ij} — coefficient from our model.

Similarly to CPU, screen power consumption can be calculated by multiplying the time screen was turned on at different brightness levels to model weight coefficients and summing the results. These coefficients are obtained by interpolating power values from the `power_profile.xml` file for minimum and maximum brightness. Timing information is obtained from `batterystats` log, which contains screen brightness change events. Each event includes time since last `batterystats` reset in milliseconds and one of the five brightness levels: dark, dim, medium, light, bright.

D. Framework execution trace and component usage statistics acquisition

After all the tests are completed, execution logs and component status are downloaded from the test device. As `logcat` logs contain a lot of additional information, but each entry type structure is regular, pattern matching using regular expressions is used to extract actual data. Then thread execution traces are separated from one another, and the obtained data is aggregated into a JSON files. These files are then processed by high-level tools based on *Navitas Framework*, but we also keep the file format open if a user wants to process the data with some custom tool.

E. Navitas Profiler implementation

Navitas Profiler is a plugin for Android Studio based on *Navitas Framework* tasks and developed to provide a practitioner a high-level tool to profile energy consumption of Android applications during the development process and visualize power consumption based on profiling results while being integrated with the IDE itself.

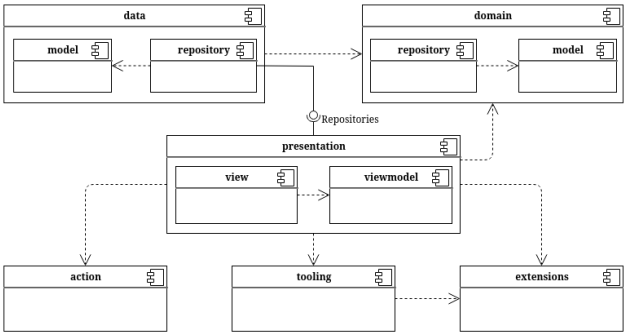

Fig. 2. *Navitas Profiler* architecture

The plugin adheres to a three-layer architecture (Fig. 2):

- *domain* — business logic, energy modelling utilities and energy profile management tools;
- *data* — an abstraction layer for *Navitas Framework* data acquisition;
- *presentation* — user interface classes.

To integrate *Navitas Profiler* to Android Studio project, its *build.gradle* file should be modified accordingly.

A common use case for *Navitas Profiler* is the following. User selects an Android module and its tests to run for profiling, as well as device energy profile — our weight coefficients for a particular test device. Then profiling is initiated on a real hardware device, and upon its completion

analyzer module receives report file in JSON format. The energy data is calculated for each method of each thread in the execution trace using the formulas discussed in Section 4.C. Note that Android API calls and third-party libraries are not profiled as no instrumentation is done for them. Then in a profiler window user can select the desired test and get detailed information about the energy consumption of all methods of the test (Fig.3).

A user can get energy consumption data for a specific method of a specific thread by selecting it in the tree view under the chart. The power readings during this method execution will be shown. Note that nested invocations will also be shown as a tree (Fig.4). Filtering execution trace by a specific thread ID is also available.

V. Experiments

We first confirmed that *Navitas Framework* is working as intended by accident. While studying energy impact of Android compiler optimizations under parallel line of work, we have found that the same code running in a separate thread using a separate class inherited from `Runnable` interface consumes a different amount of energy compared to the case where an anonymous class was used (see listings 1 and 2).

```
class TaskClass : Runnable {

    @Override
    public void run() {...}
}
...
new Thread(new TaskClass()).start();
...
```

Listing 1. Runnable as a separate class

```
...
new Thread(new Runnable() {

    public void run() {...}

}).start();
...
```

Listing 2. Runnable as an anonymous class

A test was created which ran some constant payload either under `Runnable` as a separate class, or as an anonymous class for 100 000 times. Test device was prepared according to guidelines we outlined in SLR [5]. In particular, Wi-Fi, Bluetooth and GPS were turned off, all unnecessary applications were removed or stopped, smartphone was put steady on a table and was not moved until all the test runs were over, there was a pause of 1 minute for a CPU to cool down, and CPU DVFS governor was set to `performance` for a constant maximum frequency. Thread affinity was not set for application threads. *Navitas Profiler* was utilized to measure energy consumption in both cases, and the result is shown in table I. On average, anonymous class case consumes less energy, however due to OS interference and thread scheduling it is possible some particular test runs for the first payload are lighter than for the second one. It was confirmed by disassemblying the APK that in the first case additional instructions were generated. We conclude that multiple test runs and results averaging should be incorporated into every experimental methodology concerning energy consumption.

TABLE I
Runnable with separate and anonymous class energy consumption

Test run	Separate class energy (J)	Anonymous class energy (J)
1	15.93	11.56
2	12.55	13.11
3	12.27	11.56
4	12.27	13.25
5	11.99	12.27
6	13.68	11.99
7	12.13	11.84
8	11.84	11.42
9	11.56	11.7
10	12.83	11.7
11	12.13	11.99
12	11.84	12.13
13	11.84	11.84
14	11.84	11.7
15	12.55	12.55
Average	12.48	12.04

TABLE II
Difference in energy consumption between sorting and non-srting payload

Test run	Sorting payloadg energy (J)	Non-sorting payload energy (J)
1	17.56	13.20
2	14.61	13.41
3	21.48	13.52
4	13.74	10.80
5	13.30	13.41
6	15.16	12.21
7	17.78	15.92
8	14.50	12.10
9	13.20	14.50
10	15.81	14.83
Average	15.71	13.39

In a second experiment `Runnable` creation was uniform, but the payload was different. In one case pseudo-random number generator was used with the constant seed to generate an array of 10^6 integer numbers which was then saved to internal storage. In another case array sorting was done before it was saved. Therefore second payload would consume more energy than the first one. While the experimental design deiberately favours the first payload energy-wise, such energy issues may happen in real applications when the data is sorted when it's not required do so. Table II, Fig.5 and Fig.6 show that *Navitas Profiler* is able to detect these discrepancies in both in total and individual thread energy consumption.

VI. Conclusions and Future Work

Both *Navitas Profiler* and *Navitas Framework* source codes are available at `https://github.com/Stanislav-Sartasov/Navitas-Framework`. We would like to address the following issues in our future work:

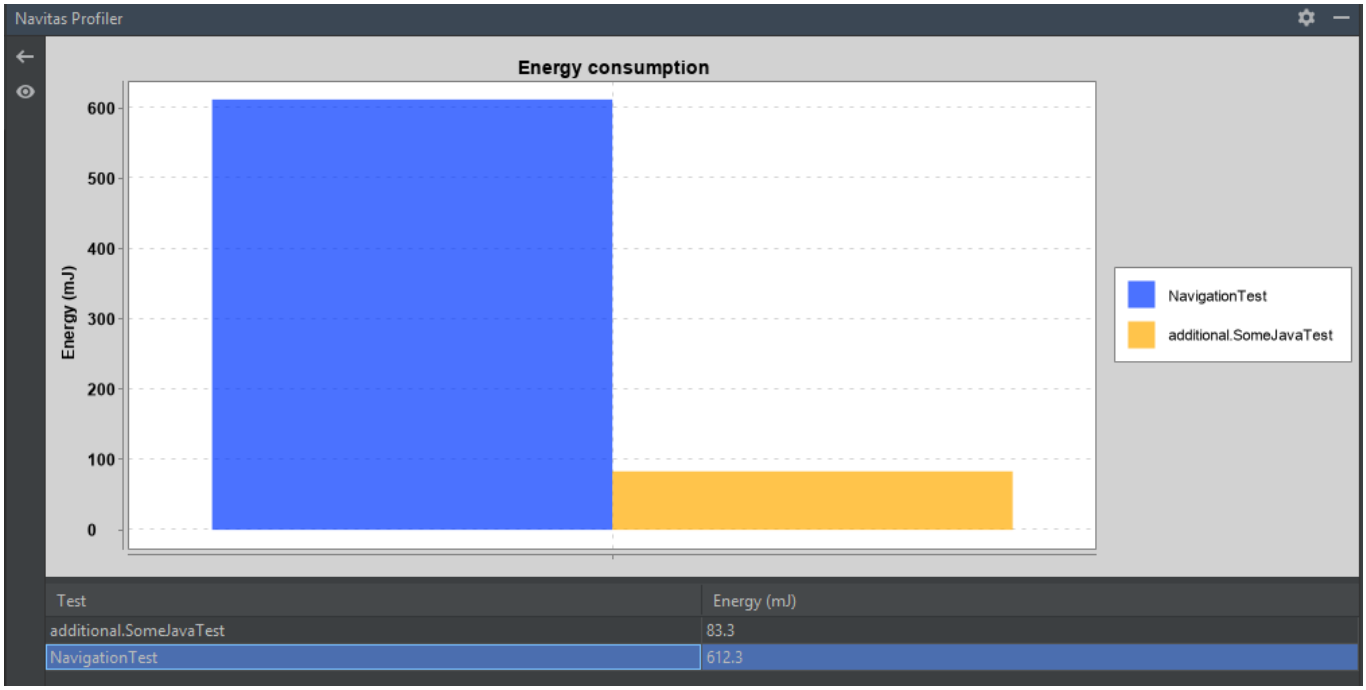

Fig. 3. *Navitas Profiler* tests energy consumption

- Currently the list of supported peripherals is very short, and we're working on expanding it with Wi-Fi, Bluetooth and GPS being the main priority.
- While currently every method of user code is instrumented, sometimes such level of instrumentation is excessive. Sometimes only the energy consumed by the entire test is required, so instrumentation might be more lightweight. An adjustment model should also be introduced for instrumented instructions to offset their own energy consumption.
- Running the same test multiple times automatically and averaging the results is a natural extension to the current testing process.
- Execution trace needs to properly support exception handling as method beginning might not have a corresponding method ending in a `logcat` output.
- Multi-threading energy consumption analysis is currently done at a pretty basic level. As we have outlined, understanding what is the energy consumption of a specific thread in an interleaved execution is a non-trivial task (see Section II.C), so adding `ftrace` execution data to our model as well as accounting for varying CPU power consumption under varying payload is a challenging research project.
- We use the data in `power_profile.xml` as a basis for our model, but actual device might be inconsistent with it [6]. Currently our model is as precise as those values, so implementing a model calibration algorithm is an important next step. Testing the quality of our model against direct measurements is another mandatory direction.

However even in its current state our tool estimates peripheral energy consumption instead of battery drain and uses Joules instead of relative units, therefre we consider it to provide a valuable and concrete information on energy consumption compared to analogues. We admit that current design supports test runs and doesn't provide real-time consumption graph like Android Studio Energy Profiler, but in our view these are just two different and equally valuable ways to work with profiling applications.

Both *Navitas Profiler* and *Navitas Framework* are used in a number of research projects of Software Engineering department of Saint Petersburg State University. As they provide a practitioner a complete set of profiling tools, we consider them to be a valuable contribution to energy profiling tools on Android platform.

References

[1] Statista Inc., "Number of smartphone users worldwide from 2014 to 2023 (in billions)," https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2020, [Online; accessed 5-April-2021].

[2] C. Sahin, F. Cayci, I. Manotas, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*, 06 2012.

[3] L. Ardito, G. Procaccianti, M. Torchiano, and A. Vetro, "Understanding green software development: A conceptual framework," *IT PROFESSIONAL*, vol. 17, pp. 44–50, 01 2015.

[4] X. Li and J. P. Gallagher, "Fine-grained energy modeling for the source code of a mobile application," in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MOBIQUITOUS 2016. New York, NY, USA: ACM, 2016, pp. 180–189. [Online]. Available: http://doi.acm.org/10.1145/2994374.2994394

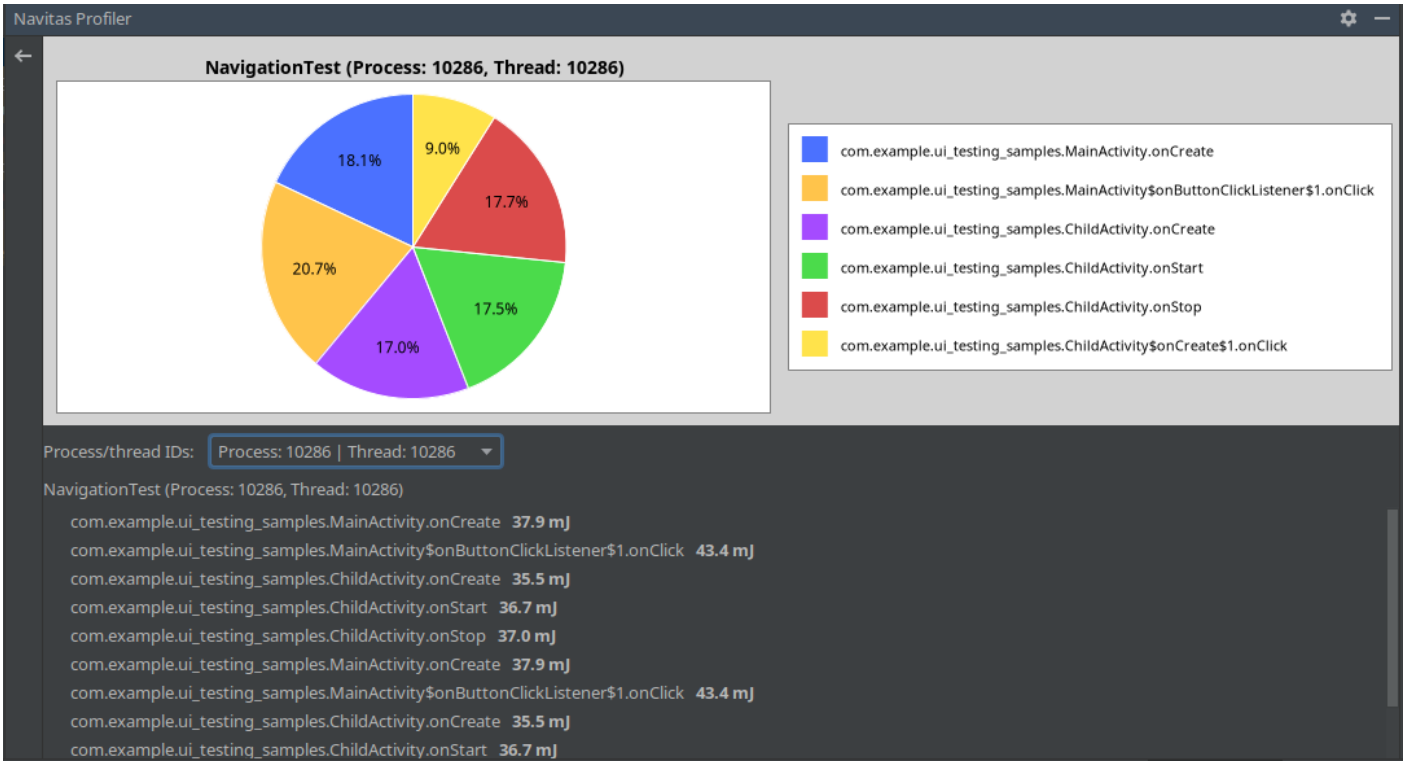

Fig. 4. *Navitas Profiler* detailed energy consumption of methods

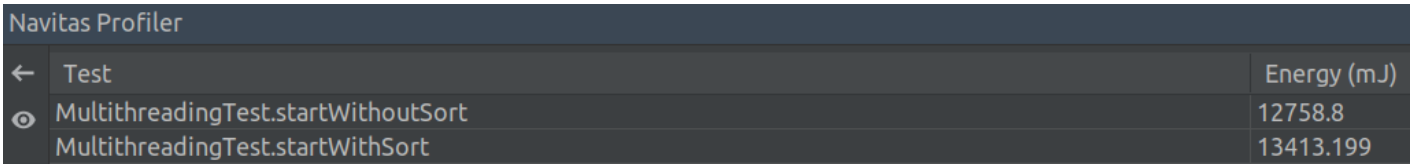

Fig. 5. *Navitas Profiler* total results for second experiment

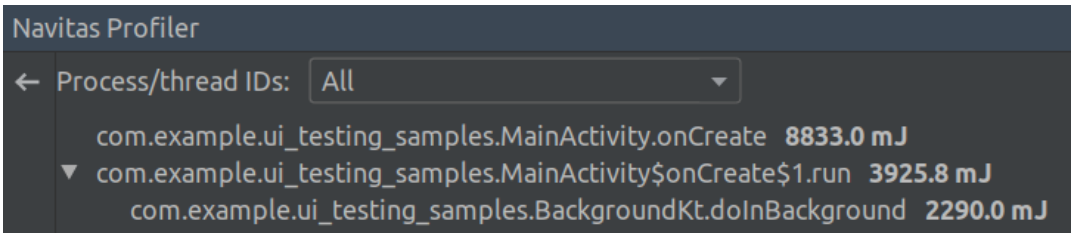

Fig. 6. *Navitas Profiler* details results for second experiment

[5] V. Myasnikov, S. Sartasov, I. Slesarev, and P. Gessen, "Energy consumption measurement frameworks for android os: A systematic literature review," in *Proceedings of the Fifth Conference on Software Engineering and Information Management 2020 (SEIM 2020)*, ser. CEUR Workshop Proceedings, 2020. [Online]. Available: http://ceur-ws.org/Vol-2691/paper10.pdf

[6] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1221. [Online]. Available: https://doi.org/10.1145/2597073.2597097

[7] C. Wilke, S. Götz, and S. Richly, "Jouleunit: A generic framework for software energy profiling and testing," in *Proceedings of the 2013 Workshop on Green in/by Software Engineering*, ser. GIBSE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 914. [Online]. Available: https://doi.org/10.1145/2451605.2451610

[8] Y. Chung, C. Lin, and C. King, "Aneprof: Energy profiling for android java virtual machine and applications," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, Dec 2011, pp. 372–379.

[9] M. Couto, J. Cunha, J. Fernandes, R. Pereira, and J. Saraiva, "Green-droid: A tool for analysing power consumption in the android ecosystem," 11 2015, pp. 73–78.

[10] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: A software-based tool for estimating the energy profile of android applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017,

pp. 3–6.
[11] S. Tsao, C. Kao, I. Suat, Y. Kuo, Y. Chang, and C. Yu, "Powermemo: A power profiling tool for mobile devices in an emulated wireless environment," in *2012 International Symposium on System on Chip (SoC)*, Oct 2012, pp. 1–5.
[12] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, June 2012, pp. 1–7.
[13] Donghwa Shin, Kitae Kim, Naehyuck Chang, Woojoo Lee, Yanzhi Wang, Qing Xie, and M. Pedram, "Online estimation of the remaining energy capacity in mobile systems considering system-wide power consumption and battery characteristics," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013, pp. 59–64.
[14] T. Reddy, *Linden's Handbook of Batteries, 4th Edition*. McGraw-Hill Education, 2010. [Online]. Available: https://books.google.ru/books?id=MXzwCfmoihYC
[15] Google Inc. (2018) Profile battery usage with batterystats and battery historian. [Online]. Available: "https://developer.android.com/studio/profile/battery-historian"
[16] "Inspect energy use with Energy Profiler," https://developer.android.com/studio/profile/energy-profiler, 2019, [Online; accessed 3-April-2021].
[17] Arm Holdings, "big.little," https://developer.arm.com/technologies/big-little, 2019, [Online; accessed 3-April-2021].
[18] "ftrace - Function Tracer," https://www.kernel.org/doc/Documentation/trace/ftrace.txt, 2008, [Online; accessed 3-April-2021].
[19] Google, Inc., "Profile battery usage with batterystats and battery historian," https://developer.android.com/studio/profile/battery-historian, 2019, [Online; accessed 3-April-2021].
[20] Monsoon Solutions, Inc., "High voltage power monitor," https://www.msoon.com/online-store, 2019, [Online; accessed 3-April-2021].
[21] "[app][5.0+] ampere the charging meter," https://forum.xda-developers.com/android/apps-games/app-ampere-charging-meter-t3012890, 2019, [Online; accessed 3-April-2021].
[22] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855861
[23] Shigeru Chiba, "javassist by jboss-javassist," https://www.javassist.org/, 2019, [Online; accessed 3-April-2021].
[24] The Apache Software Foundation, "Byte-code engineering library," https://commons.apache.org/proper/commons-bcel/, 2019, [Online; accessed 5-April-2021].
[25] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, 08 2012.
[26] Google, Inc., "Android studio," https://developer.android.com/studio, 2019, [Online; accessed 3-April-2021].
[27] ——, "Processes and threads overview," https://developer.android.com/guide/components/processes-and-threads, 2019, [Online; accessed 5-April-2021].
[28] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7 – 15, 2009, special Section - Most Cited Articles in 2002 and Regular Research Papers. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908001390
[29] "Energy consumption on Android Studio Profiler," https://stackoverflow.com/questions/52647045/energy-consumption-on-android-studio-profiler, 2018, [Online; accessed 3-April-2021].