A Software-based Approach for Source-line Level Energy Estimates and Hardware Usage Accounting on Android

Alexandre Cornet and Anandha Gopalan
Department of Computing, Imperial College London, London, SW7 2AZ, UK
Email: alex.cornet.ac@gmail.com, a.gopalan@imperial.ac.uk

***Abstract*—As users rely more on their mobile devices, energy inefficient software is a real threat to user experience. Early tools for developers focussed on expensive power measurement hardware and software-based approaches were introduced to relieve them of such requirements. These tools highlighted the most energy-inefficient parts of the code, but the developer still had to find and understand the exact causes of energy drain. Also, there was no mapping of hardware energy activity to code and no accounting for tail energy. To this end, this work focusses on providing source-line level energy estimates and maps the drain caused by Wi-Fi back to the code while accounting for tail-energy.**

Keywords–Green-computing, Tail-energy, Energy profiling

I. MOTIVATION

Mobile technologies have become ever-present in our daily lives and many people's personal and professional interactions now depend on their smartphone or tablet. The power consumption of mobile devices obviously grew with the duration of their usage and the complexity of hardware and software they involve. By nature, these devices are used away from power sources and battery has thus become a critical component for the user experience. However, battery technology hasn't managed to keep up with these requirements and hence energy efficiency has become a major concern of users who are now looking for feedback to understand how applications drain the battery of their devices [1]. Research even shows that implementations perceived as energy greedy will receive lower ratings from users [2]. The ability to build energy efficient software consequently rewards developers with a competitive advantage on the market. However, energy optimisation is often counter intuitive to many developers and there is no real guidance. For example, there is no clear correlation between energy and time efficiency [3], and time optimisation is thus not always useful to reduce the energy footprint of a software. Also, some popular design principles, such as the decorator pattern have bad energy efficiency [4].

Researchers have consequently developed tools to provide guidance to developers by profiling the energy drain of their code. These tools were initially tied to specific and expensive power measurement platforms which inherently limited their use. Software-based approaches were introduced so that energy profiling techniques are accessible to the vast majority of developers by placing no hardware requirements on them. The first key contribution of this work is to provide source-line level energy estimates to the developers, which is achieved by extending Orka [5] [6].

To provide accurate and meaningful energy feedback, however, energy profilers should also take into account the drain caused by various hardware components. This means being able to map the hardware energy usage back to the code with the finest granularity possible. Moreover, mobile devices exhibit several asynchronous power behaviours, the most significant of which is *tail-energy*, which proves challenging for the development of energy profilers [7] [8]. A component exhibits tail-power behaviour if it stays in a high power for a constant time after processing a workflow and this phenomenon can cause significant energy drain. Thus far, only a few contributions, such as *eprof* [7] have focussed on tail-energy by relying on hardware- or model-based approaches. Therefore, no tool accounting for tail-energy was accessible to the majority of developers. To allow for this, the second major contribution of this work is for our solution to also provide tail-energy accounting. We have focussed exclusively on Wi-Fi for now and will in future include other hardware components.

The remainder of this paper is organised as follows: Section II outlines related work, Sections III and IV detail the major contributions. Section V provides an evaluation, and Section VI concludes the paper and provides ideas for future work.

II. BACKGROUND RESEARCH

This section outlines other works related to this paper.

A. High level guidance

Software optimisation has tended to focus on time, but according to [3], choosing an energy-efficient implementation over a time-efficient implementation allowed more operations to be performed on a mobile device. [9] reviewed a set of coding practices and proved that reducing memory usage has a low impact on reducing the energy usage of code. Related conclusions were obtained in [10] as they proved code obfuscation negatively impacts the energy efficiency of the underlying software. In [7], they profiled and analysed the energy footprint of six of the top-ten applications on the *Google Play Store*. They found that most energy is spent in I/O and that most of this energy is due to tail-behaviour. The authors in [11] surveyed 55 applications in order to mine energy greedy calls to the Android API. They obtained the cost of 800 API calls and found that the Android API was the most energy inefficient part of applications. These findings are however, limited by the fact that part of the energy-greedy I/O activities are handled in the Java API and therefore don't appear in this study. Finally, tail-behaviour was not considered.

High level guidance provides a first mean to produce more energy-efficient software. However, these findings don't allow developers to know which specific parts of code consume the most energy. Researchers therefore focussed on developing energy profilers able to provide detailed feedback about the energy footprint of applications.

B. Hardware-based profilers

Hardware-based energy profilers such as *PowerScope* [12] and *GreenMiner* [13] involve the use of power measurement platforms, i.e., mobile devices with embedded power sensors monitoring the energy drain caused by running processes. *vLens*, an energy-profiler combining program analysis and statistical modelling to provide energy usage information at the source line level was introduced in [8]. These approaches

 ISBN: 978-1-61208-635-4

however, are tied to expensive power measurement platforms, involving an overhead for their users.

C. Model-based profilers

Model-based solutions build power models of mobile devices, which are further reused away from measurement platforms to profile the energy footprint of applications. *eCalc* [14] assumes the availability of a CPU power profile containing the energy drain associated with each CPU instruction in order to provide drain caused by the CPU at the method granularity. *PowerBooter* uses battery voltage sensors and the knowledge of the battery behaviour to automate the power model generation and *PowerTutor* further uses this model to compute the energy costs [15]. *TailEnder* provides models focussing on the drain of I/O components, namely 3G and GSM. *eprof* [7] focusses on mapping the drain caused by hardware components while accounting for entities such as processes, threads and routines. For each entity, *eprof* produces an energy tuple `(utilisation_draw, tail_mode_draw)` and is therefore one of the few contributions accounting for tail energy and comparing it to utilisation drain. They also present methods to generate these power models by correlating certain behaviours with specific power states and identifying these behaviours as trigger conditions on the finite state machine [16]. Model-based techniques suffer one major drawback: the underlying power models are specific to the profiler and are therefore not widespread nor publicly available.

D. Software-based profilers

Software-based solutions were introduced to allow for portable and widely accessible energy profilers. They do not require any knowledge of the device behaviour or access to specific hardware. Users usually provide their application alongside a test scenario. Based on the findings in [11], [2] introduced *EcoDroid*, an energy profiler that focuses on ranking applications instead of providing energy estimates. The Power Estimation Tool for Android Application (*PETrA*) uses new tools of the Android Open Source project focussing specifically on energy profiling [17]. *PETrA* simulates a typical execution of the tested application on an actual device using a user-provided script. The execution trace is recorded using `dmtracedump` and the `batterystats` history is collected using `dumpsys`. *PETrA* finally replays these files to compute the battery drain during the execution of each method and provides the user with the energy estimates at the method level.

Orka is one of the first software-based energy profilers [5]. It takes an Android application and a dynamically created execution trace to provide energy usage estimations as feedback. The energy cost of a routine is calculated using the energy costs of the Android API calls (from [11]). We had access to the source code of *Orka* and its design and assumptions were compatible with our project goals. Hence, extending *Orka* proved to be the most viable solution. The basic workflow of *Orka*, shown in Figure 1, is divided into three parts: (i) application is instrumented to log the API calls, (ii) instrumented application is then run on the Android emulator using a user-supplied `monkeyrunner` script, and (iii) execution traces are analysed and results are presented to the user. The execution analysis is done by using the logs and `batterystats` data and is used to calculate the total cost of a routine which is obtaining using: $cost(routine) = \sum_{API \in routine} cost(API)$. The `batterystats` file also gives the breakdown by component of the hardware energy usage.

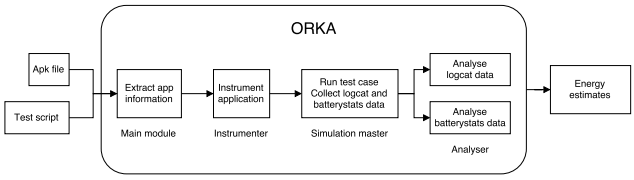

Figure 1. Basic workflow of *Orka*

III. Providing source-line level energy estimates

The key extension to Orka is to account for API calls occurring in subroutines and provide this at the source-line level rather than the method-level as was the case previously. To this end, *Orka* now maintains an in-memory version of the call stack while it replays the logs and attributes any API calls to all the routines in the call stack. Therefore, *Orka* tallies the calls occurring *during* the execution of this routine, possibly in a subroutine with the results displayed as a percentage of the total routine cost. Taking control-flow into account, the cost of a line of code is defined as the cost of a single execution of this line multiplied by the average number of times it was executed per call. *Orka* approximates the energy usage of a routine with the cost of its calls to the Android API and assumes that remaining parts of the code have a marginal energy footprint. On top of tallying the number of API calls during a routine execution, *Orka* uses the information provided by the `smali .line` instruction to know where in the source code each of these calls occurred. Since all the API calls happening during the execution of a subroutine are now attributed to the parent routine as well, for a given routine, all API calls happening during the execution of a subroutine will be attributed to the line from which the subroutine is invoked.

A. Implementation

1) Injector: The first step was to allow the instrumented application to log `(apiName, lineNumber)` pairs. On finding the `smali .line` statement, the injector now updates a variable containing the source line number corresponding to the current instruction. On finding an API invocation, this information is then appended to the API's name and passed to the `APILog` function. At this point, *Orka* is able to know where the API was called in a method's body and to detect when the API was called from a subroutine. In order to map this cost to a specific line in the parent routine, we have to allow the instrumented application to log messages indicating subroutine invocations alongside the line from which the subroutine was called. On finding an `invoke` statement, the injector now checks whether the called routine is user-defined and injects a call to a logging function if so. To enforce one main requirement of the injector, namely to ensure minimum code is injected, log messages shouldn't be inserted before the invocation non-injected routines, as this information won't be useful during the analysis. Therefore, the injector needs to know which routines are injected prior to the injection and add log messages only before calls to injected subroutines. To this end, a preparatory phase was added to the injector work-flow: injected files are now scanned prior to the injection in order to build the set of injected methods.

2) Analyser and `routine` class: The `routine` class calculates the cost of a routine as the total cost of all its API calls. To implement fine-grained energy feedback, we instead identify the cost of a routine as the sum cost of all its lines. The cost of a line is defined as the cost of the API calls occurring on

```
method RecList.onCreate, Average cost:
    0.0197754308, Calls: 1.0
      2.64% l-1
      1.45% l192     calling RecList.
          InitializeRepeatButton
      0.88% l227     calling RecList.
          InitializeThemedButtonsBackgrounds
      0.57% l229     calling Preferences.
          getCurrentlyPlayingFilePath
     29.04% l247     calling RecList.
          updateSongList_extended
     21.80% l257     calling android.os.Bundle.
          getString
     21.80% l258     calling android.os.Bundle.
          getString
     21.80% l260     calling android.os.Bundle.
          getString
```

Figure 2. Example of reconstructed source code with fine-grained guidance

that line. The `routine` class was thus extended to generate a reconstructed source code including energy estimates from the API data – typical output shown in Figure 2. To store the line from which a subroutine is called, a second stack was added alongside the call stack. On finding a subroutine invocation log, the line number from which the subroutine is called is added to the stack. On finding an exiting statement, the last line number is popped from the stack. By merging these two stacks, we obtain a stack of pairs `(routineName, lineNumber)` indicating which line the API calls should be attributed to in a given routine. Finally, the names of subroutines are stored in a separate dictionary, the keys of which are line numbers. In some specific cases, e.g., a class accessing a static variable of another class, injected constructors are called implicitly. As *Orka* is not yet able to detect such cases, subroutine invocation statements are sometimes missing and there is no information regarding the line number corresponding to the call. However, by comparing the two stack sizes we are able to detect this and a default value is added to the stack to indicate the absence of any information.

IV. Providing Hardware Usage Accounting

In order to account for tail-energy, we use Wi-Fi, a frequently used component. For a given routine, the drain associated to each energy state (active, tail, idle) of the Wi-Fi antenna can be easily computed by multiplying the time spent in this state by the corresponding power drain provided by the power state machine. Therefore, focussing on the time spent in each state rather than the corresponding energy drain would work fine. For each routine, *Orka* should ultimately be able to provide an estimate of the drain caused by Wi-Fi alongside an energy tuple. The main aim is to correlate the Wi-Fi energy activity with routine calls, therefore *Orka* needs to know when routine invocations start and end, and have access to the power drain caused by Wi-Fi at any time. The start-time and end-time of any routine call could be easily obtained by leveraging the `logcat` enter and exit statements and by using an output mode of `logcat` including the timestamps of log messages.

A. Monitoring the power consumption of Wi-Fi

Just like *eprof*, *Orka* needs access to the power consumption of Wi-Fi with the highest sampling period possible. However, to enforce the software-based nature of *Orka*, power-measurement platforms and complex energy models need to be replaced with information provided by the operating system.

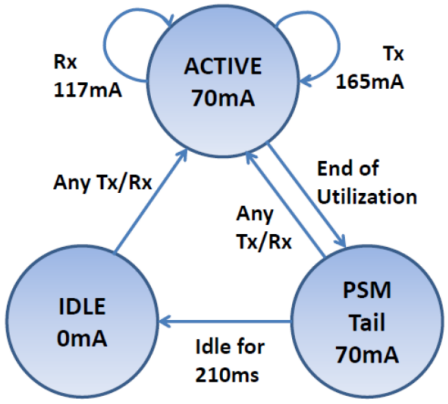

Figure 3. Typical power state machine of a Wi-Fi antenna [19]

As the power behaviour of Wi-Fi antennas can be accurately described by a power state machine, *Orka* could access the power consumption caused by Wi-Fi at any time by monitoring in which energy state the antenna is.

1) Wi-Fi power state machine: The power behaviour of most hardware components, including Wi-Fi, can be accurately described using a power state machine. As shown in Figure 3, Wi-Fi antennas exhibit three power states: active, tail, and idle. By definition, the bandwidth in idle and tail mode, as well as the drain in idle mode are always zero. The bandwidth and drain in active mode, the tail-time and the tail drain differ from one antenna to another. As *Orka* only focusses on energy and not workload, the bandwidth in active mode is not relevant to this work. To this end, the power drain in active mode for a specific device can be obtained using the power profile. From *Android* 5.0, devices come with a power profile providing the energy drain caused by each hardware component in its various energy states. While this includes the drain in active mode for the various components, we found that the power profile doesn't include the drain caused by a component in tail mode and hence is not useful to accurately compute the energy due to tail-behaviour. However, we found in the literature that the drain in tail-mode can be approximated by half the drain in active mode [18] and *Orka* uses this approximation. Finally, we investigated whether we could get the the exact value of the tail-time for any Wi-Fi antenna. Unfortunately, techniques such as the one in [19] involve power-measurement platforms. We hence approximated this value with the one shown in Figure 3. Due to these approximations, *Orka* has access to a complete state machine describing the power behaviour of the Wi-Fi antenna. To monitor the power drain caused by Wi-Fi, *Orka* only needs to know in which power state the antenna is.

2) Monitoring the energy-state: Based on [17], we looked at using the energy tools from the Android Open Source Project. The `batterystats` history contains a timeline of energy related events since the last charge or reset, but it doesn't record switches to the tail state. Hence, we had to use a new approach – monitor events which trigger these switches, namely network traffic events. The Dalvik Debug Monitor Server (DDMS) provides tools to monitor the network traffic in real time for a given application [20]. Using this, we found that network statistics at the application level are stored in `proc/net/xt_qtaguid/stats` [21]. This file contains one line per `(app_uid, tag)` pair, describing the associated network traffic, as shown in Figure 4. A `Python`

```
idx iface uid_tag_int cnt_set rx_bytes
    rx_packets tx_bytes tx_packets
2 wlan0 0 0 200888 1096 79636 888
3 wlan0 0 1 0 0 0 0
```

Figure 4. Extract of `proc/net/xt_qtaguid/stats`

procedure was used to parse this file and return a pair `(rx_bytes, tx_bytes)`, aggregating all the incoming and outgoing traffic induced by a given application. The command `adb shell cat` was used to fetch this file and was able to update the statistics once every 45ms. Network traffic caused by an application with a sampling period of about 50ms could therefore be potentially monitored using an actual device as there is no Wi-Fi emulation on AVDs.

B. Implementation

Using these findings, two pieces of software were implemented in order to correlate the routine calls with the Wi-Fi energy activity; (i) a `Python` script to log the switches between the various energy states of the Wi-Fi antenna, and (ii) an analyser to process this data and generate results interpretable by the user. A test application was also written to evaluate the quality of the results generated.

1) Monitoring the Wi-Fi energy state: The pseudo-code in Figure 5 fetches the most recent network data and compares it to a previous one. Any change in this data indicates that network traffic was induced by the application and the antenna switched to active mode. Otherwise, no traffic occurred and if the antenna was in active mode, it switches to tail mode and resets a counter indicating the time when tail mode started. If the antenna was in tail mode, this counter is used to compute the time spent in that mode and switch to idle mode if needed. If the antenna was already in idle mode, it remains so. We ensure that, despite the loss of precision due to the sampling period, the duration of tail mode is never longer than the tail time specified by the transition condition in the state machine.

In order to correlate the routine calls and the energy consumption of the Wi-Fi antenna, *Orka* needs to merge-sort these two files and replay the resulting log file. To accurately sort the timestamps, the clocks used in both logs need to be synchronised. Many networking protocols able to synchronise clocks are available, but most of them are quite complex to use. As *Orka* has to deal with the Android clock, which is used in the `logcat` dump, this clock should also be used to timestamp the switches between the energy states. To this end, the host machine opens the connection with the device and send two commands: the first one to get the epoch time and the second to fetch the file containing the network statistics.

The logs generated by the `networkMonitor` module (Figure 6) will be referred to as `netstats` logs or traces. Comparing these results with those produced by DDMS, we found that the antenna was successfully logged in active or tail mode when traffic was reported by DDMS. Moreover, the sampling period of *Orka* is twice as small as the one of DDMS, as the smallest sampling period offered by this tool is 100ms.

2) Analysing the traffic and execution traces: Once the simulation terminates, *Orka* needs to process this new data to compute the estimates modelling the tail-behaviour induced by each injected routine. To achieve this, the `routine` class is extended to include a dictionary, which stores the time spent by the Wi-Fi antenna in each energy state while the routine was executed. For e.g., a routine executed for 10s without using Wi-Fi will be modelled as: {`'ACTIVE': 0.0, 'TAIL': 0.0, 'IDLE': 10.0`}. The `logcat` data is used to keep track of the call stack and the `netstats` data to update the energy state of the Wi-Fi antenna. The main function parses these two traces and performs a merge sort so as to process the logs chronologically. For each new entry, the time elapsed in the current state since the last entry is added to all the methods in the call stack and the Wi-Fi state and the call stack are then appropriately updated. Once the logs have been fully processed, the results, which include the tuples storing the time spent in each energy state for all the injected routines, are written to disk. Figure 7 shows an example extract.

Input: Active ADB connection to an actual device
Output: Logs of the energy states of the Wi-Fi antenna

Get first network statistics S_0 at current time t_0;
$state \leftarrow$ `IDLE`;
while *True* **do**
  Get network statistics S_1 at current time t_1;
  if $S_0 \neq S_1$ **then**
    $state \leftarrow$ `ACTIVE`;
  else if *state* = `ACTIVE` **then**
    $state \leftarrow$ `TAIL`;
    $tail_{start} \leftarrow t_0$;
  end
  if $state =$ `TAIL` **and** $t_1 - tail_{start} \geq tail_{time}$ **then**
    Log $(t_0, state)$;
    $t_0 \leftarrow tail_{start} + tail_{time}$;
    $state \leftarrow$ `IDLE`;
  end
  Log $(t_0, state)$;
  $t_0 \leftarrow t_1$;
  $S_0 \leftarrow S_1$;
end

Figure 5. Network monitor

```
1503657227.287407 ACTIVE
1503657227.323574 TAIL
1503657227.359771 TAIL
1503657227.394717 TAIL
1503657227.428777 ACTIVE
1503657227.467203 TAIL
1503657227.518874 ACTIVE
1503657227.589833 ACTIVE
1503657227.644556 TAIL
1503657227.681700 TAIL
1503657227.716493 TAIL
1503657227.753575 TAIL
1503657227.789498 TAIL
1503657227.829724 TAIL
1503657227.864556 IDLE
1503657227.882524 IDLE
```

Figure 6. Extract of a typical `netstats` output

Building on these new modules, *Orka* is now able to correlate the network activity with the routine calls and attribute the energy usage to them, while taking tail-energy into account.

V. Evaluation

In order to get an insight of the accuracy of the results generated by the implementation, the decision was made to test it against an application designed specifically for this purpose.

1) High-level expectations: As outlined by [7], the power behaviour of the Wi-Fi mostly depends on the total network traffic and on the density of this traffic. In the case when the traffic is particularly dense, the energy drain should be roughly proportional to the total traffic since the antenna will remain

```
MainActivity.<init> {'ACTIVE': 0.0, 'IDLE':
    0.011000015640258789, 'TAIL': 0.0}
MainActivity$1.onCheckedChanged{'ACTIVE': 0.0,
    'IDLE': 0.030999989891052246, 'TAIL': 0.0}
MainActivity$2.onClick {'ACTIVE': 0.0, 'IDLE':
    0.016999959945678711, 'TAIL': 0.0}
MainActivity.onCreate {'ACTIVE': 0.0, 'IDLE':
    0.13199996948242188, 'TAIL': 0.0}
MainActivity$SendGet.run {'ACTIVE':
    1.5723857879638672, 'IDLE':
    0.8458666801452637, 'TAIL':
    1.626746654510498}
```

Figure 7. Extract of a typical output of the `networkAnalyser`

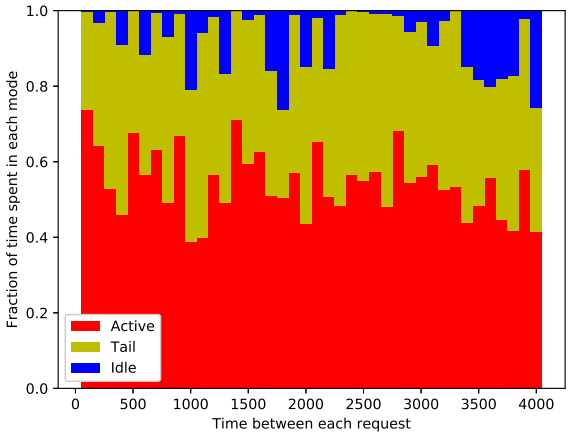

Figure 8. For the method `SendGet.run()`

in active mode and won't exhibit any tail-energy behaviour. However, in the situation when the antenna has to process a small work-flow (sparse traffic) as soon as it enters its idle state, the antenna will spend most of its time in the tail mode. Based on this, we built a test application which allows to generate traffic of various densities and to check whether the results generated by *Orka* are fitting with these principles.

2) Designing a test application: An Android application was created using Android Studio to simply send a `HTTP GET` request every `T` milliseconds to a constant target URL. `T` was initially set to 1000ms and the target URL to `http://www.google.com`. A simple GUI was added later to let the user specify the target URL and the parameter `T`.

3) Running the tests: *Orka* was then run on this test application using the target URL `http://www.google.com` for various values of `T` between 100ms and 4000ms in steps of 100ms. To this end, `monkeyrunner` scripts were then generated in order to automatically set the right value of `T` and let *Orka* monitor the traffic during 20 seconds. The results generated for $\texttt{T} = 1000ms$ are presented in Figure 7.

4) Analysing the results: At first glance, it was found that only the method `SendGet.run()`, which fires the `HTTP` request was attributed significant network usage, i.e. time in active and tail mode. All other methods were only attributed time in idle mode. This shows that *Orka* was able to detect which method was producing network traffic, and therefore to map the hardware energy usage back to the code. Based on this, it is relevant to compare the energy-state tuples of the method `SendGet.run()` for all values of `T` and to check whether the expectations described in Section V-1 were met.

Figure 8 shows the fraction of the time spent in each mode

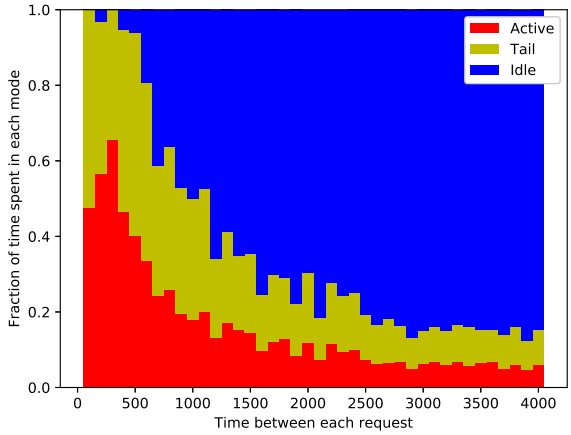

Figure 9. At the application level

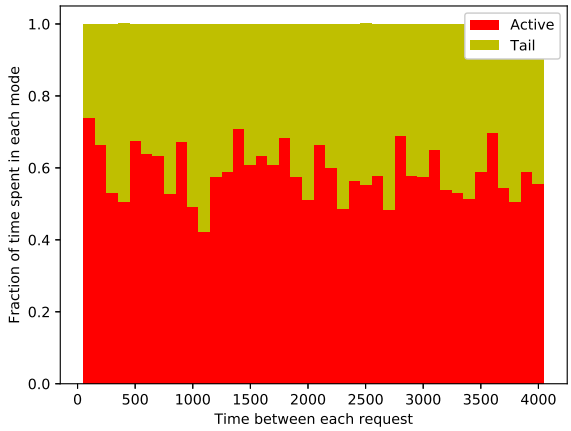

Figure 10. For `SendGet.run()`, ignoring idle mode

depending on the time `T` between each request. It seems that these ratios are constant with respect to the time between each request but include a significant amount of noise. This result may seem surprising at first, as one would expect the fraction of time spent in idle mode to increase with `T`, while the fraction of time spent in tail mode to decrease with `T`. This can be explained by the fact that to compute the energy tuple of a routine, the analyser only focusses on Wi-Fi activity during the execution of this routine. Nevertheless, by definition, tail energy continues beyond the execution of the routine and *Orka* is therefore not accounting for part of the tail energy and of time spent in idle mode. In order to improve the accuracy of the results at the method level, *Orka* needs to implement more complex accounting policies, such as the last-trigger accounting policy, where the routine which last triggered a hardware component will be attributed the energy consumption that follows until another routine accesses this component.

To understand the Wi-Fi activity not attributed to any routine by *Orka*, we looked at the energy tuples at the application level. Figure 9 shows the fraction of the time spent in each mode depending on `T` at the application level. As expected, the fraction of time spent in idle mode increases as the traffic gets less dense but this graph doesn't allow us to draw conclusions about the active and tail modes, although it seems that the fraction of time spent in active and tail modes are similar.

 ISBN: 978-1-61208-635-4

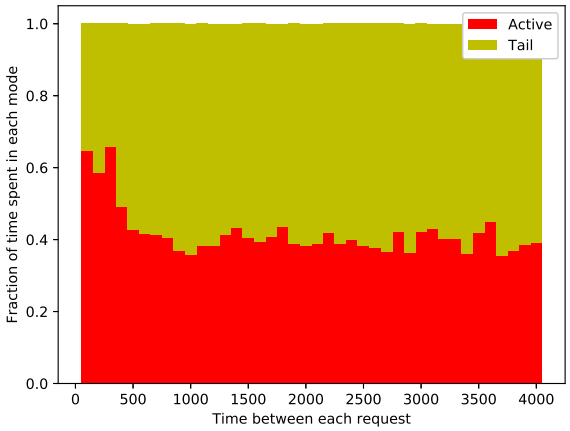

Figure 11. At the application level, ignoring idle mode

Figures 10 and 11 show Figures 8 and 9 redrawn, but with a focus on time spent only in active and tail modes. As we can see, at the method level, as the fraction of time spent in idle mode was small as compared to the time spent in the other modes, Figures 8 and 10 are very similar. However, Figures 9 and 11 show these results at the application level and there *Orka* attributes more time in tail mode as the traffic gets less dense and hence meets the high-level expectations described in Section V-1. Moreover, it seems that the fraction of the time spent in tail mode quickly reaches its maximum of about 60%, for values of `T` higher than 500ms. Finally, according to these results, the Wi-Fi antenna spends at least 40% of the time in tail mode, even for a dense traffic. This would suggest that at least 20% of the drain caused by Wi-Fi is due to tail-behaviour.

VI. Conclusions and Future Work

In this paper, we presented a software-based approach to providing fine-grained energy feedback (at the source-line level) to developers, enabling them to investigate energy bugs effortlessly. This work is also able to map the energy drain caused by the Wi-Fi antenna back to the code and to partially account for the tail-energy. One of the limitations of *Orka* is its heavy reliance on the cost of Android APIs found by [11], which should be updated to have the energy estimate of newer APIs to ensure accurate feedback. Orka operates on the main assumption that the cost of the routines making no calls to the Android API is marginal. However, a routine may not make any calls to the Android API, but instead invoke a subroutine which includes these. To this end, the injector should build a call graph of the user-defined routines, the leaves and nodes of which would be respectively the API calls and the routines. Moreover, the energy estimates generated by monitoring the energy-activity of the hardware should be included in the energy estimates provided by *Orka* at the method-level. Finally, this paper focussed exclusively on Wi-Fi and should include all other hardware components.

References

[1] M. V. Heikkinen, J. K. Nurminen, T. Smura, and H. Hämmäinen, "Energy efficiency of mobile handsets: Measuring user attitudes and behavior," Telematics and Informatics, vol. 29, no. 4, 2012, pp. 387–399.

[2] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of android apps," in Proceedings of the 4^{th} International Workshop on Green and Sustainable Software, 2015, pp. 8–14.

[3] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the "best" sorting algorithm for optimal energy consumption." in ICSOFT (2), 2009, pp. 199–206.

[4] C. Sahin et al., "Initial explorations on design pattern energy usage," in Proceedings of the 1^{St} International Workshop on Green and Sustainable Software (GREENS), 2012, pp. 55–61.

[5] B. Westfield and A. Gopalan, "Orka: A New Technique to Profile the Energy Usage of Android Applications," in Proceedings of the 5^{th} International Conference on Smart Cities and Green ICT System, SMARTGREENS, 2016, pp. 213–224.

[6] "Orka source code," https://github.com/acornet/orka, retrieved: May 6^{th}, 2018.

[7] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in Proceedings of the 7^{th} ACM european conference on Computer Systems, 2012, pp. 29–42.

[8] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in Proceedings of the International Symposium on Software Testing and Analysis, 2013, pp. 78–89.

[9] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in Proceedings of the 3^{rd} International Workshop on Green and Sustainable Software, 2014, pp. 46–53.

[10] C. Sahin et al., "How does code obfuscation impact energy usage?" Journal of Software: Evolution and Process, 2016.

[11] M. Linares-Vásquez et al., "Mining energy-greedy api usage patterns in android apps: an empirical study," in Proceedings of the 11^{th} Working Conference on Mining Software Repositories, 2014, pp. 2–11.

[12] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in Proceedings of the 2^{nd} IEEE Workshop on Mobile Computing Systems and Applications, 1999, pp. 2–10.

[13] A. Hindle et al., "Greenminer: A hardware based mining software repositories software energy consumption framework," in Proceedings of the 11^{th} Working Conference on Mining Software Repositories, 2014, pp. 12–21.

[14] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in Proceedings of the 1^{St} International Workshop on Green and Sustainable Software (GREENS), 2012, pp. 1–7.

[15] L. Zhang et al., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010, pp. 105–114.

[16] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in Proceedings of the 6^{th} conference on Computer systems, 2011, pp. 153–168.

[17] D. Di Nucci et al., "Software-based energy profiling of android apps: Simple, efficient and reliable?" in Proceedings of the 24^{th} International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, 2017, pp. 103–114.

[18] "Optimizing Downloads for Efficient Network Access," https://developer.android.com/training/efficient-downloads/efficient-network-access.html, retrieved: March 31^{St}, 2018.

[19] N. Ding et al., "Characterizing and modeling the impact of wireless signal strength on smartphone battery drain," in Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, 2013, pp. 29–40.

[20] "DDMS Network traffic tool," https://developer.android.com/studio/profile/ddms.html#network, retrieved: March 31^{St}, 2018.

[21] "Stackoverflow - How does Android tracks data usage per application?" https://stackoverflow.com/questions/31455533/how-does-android-tracks-data-usage-per-application, retrieved: March 31^{St}, 2017.

 ISBN: 978-1-61208-635-4