In-vivo and offline optimisation of energy use in the presence of small energy signals – A case study on a popular Android library

Mahmoud A. Bokhari
Optimisation and Logistics, School of Computer Science, The University of Adelaide, Australia
Computer Science Department, Taibah University, Kingdom of Saudi Arabia
mahmoud.bokhari@adelaide.edu.au

Brad Alexander
Optimisation and Logistics, School of Computer Science, The University of Adelaide, Australia
bradley.alexander@adelaide.edu.au

Markus Wagner
Optimisation and Logistics, School of Computer Science, The University of Adelaide, Australia
markus.wagner@adelaide.edu.au

ABSTRACT

Energy demands of applications on mobile platforms are increasing. As a result, there has been a growing interest in optimising their energy efficiency. As mobile platforms are fast-changing, diverse and complex, the optimisation of energy use is a non-trivial task.

To date, most energy optimisation methods either use models or external meters to estimate energy use. Unfortunately, it becomes hard to build widely applicable energy models, and external meters are neither cheap nor easy to set up. To address this issue, we run application variants in-vivo on the phone and use a precise internal battery monitor to measure energy use. We describe a methodology for optimising a target application in-vivo and with application-specific models derived from the device's own internal meter based on jiffies and lines of code. We demonstrate that this process produces a significant improvement in energy efficiency with limited loss of accuracy.

CCS CONCEPTS

• **Computing methodologies** → *Genetic programming*; • **Hardware** → *Batteries*; • **Software and its engineering** → *Search-based software engineering*;

KEYWORDS

Non-functional properties, energy consumption, mobile applications, Android, multi-objective optimisation

ACM Reference Format:
Mahmoud A. Bokhari, Brad Alexander, and Markus Wagner. 2018. In-vivo and offline optimisation of energy use in the presence of small energy signals – A case study on a popular Android library. In *EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '18), November 5–7, 2018, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3286978.3287014

1 INTRODUCTION

Energy demands on mobile platforms are increasing as users spend more time on their devices [37] and their applications become more powerful. In response, hardware manufacturers have given a very high priority to improving battery capacity [38] and operating system vendors have started to ration energy-hungry resources [19]. Unfortunately, it is still the case that mobile application vendors can produce applications that use too much energy [27]. A cause of this problem is a lack of developers' skills in optimising applications for energy [30]. Search-based software engineering (SBSE) can help address this problem through automated search for energy-efficient variants of mobile software [3, 26].

Current work in automated optimisation of energy use has employed models derived from external meters to drive search [5, 7, 9, 26, 34]. However, as operating system behaviour becomes more complex and platforms become more diverse, such models are becoming less generally applicable [13]. An alternative approach to energy evaluation is to test application variants for energy use *in-vivo*, i.e., on the device itself [3] using the device's internal meter.

Contributions. In this work we use measurements from the device's own internal meter that, for the first time:

(1) build energy models that are subsequently used for successful optimisation of energy use of a CPU-bound application and, alternatively,
(2) successfully guide the same optimisation task using direct sampling of the internal meter *during* optimisation.

By using the internal meter of the phone we make progress toward optimisation processes customised to each platform and its current software environment. As hardware platforms and software configurations become more complex and diverse we envisage the ability to create custom models in this way will become increasingly important.

We also demonstrate that it is possible to overcome the relatively low accuracy and resolution of the internal meter through:

(1) simple rewrites to the code of the source application to amplify the signal the code produces whilst running in its test harness, and
(2) by performing an initial in-vivo sensitivity analysis to identify the most promising targets for optimisation within the application code.

Through these measures we demonstrate that it is possible to significantly reduce the energy use of Rebound: a short-running physics library, for animating GUI interfaces that is installed on over 1 billion devices worldwide.

MobiQuitous '18, November 5–7, 2018, New York, NY, USA

ACM ISBN 978-1-4503-6093-7/18/11.
https://doi.org/10.1145/3286978.3287014

The rest of the paper is structured as follows. After putting our work into the context of existing work in the next section, we present our experimental methodology and describe our preliminary experiments in Section 3. Section 4 describes the energy optimisation experiments, their results and their validation. Finally, we present our conclusions in Section 5.

2 RELATED WORK

Related work can be divided into work that builds energy models for CPUs on mobile devices and work that performs automatic energy optimisation on code.

In terms of CPU energy models, many works have used an external meter to help derive energy models [11, 17, 22, 31]. Such meters, while accurate, are increasingly difficult and expensive to set up, among other, because batteries are often non-removable nowadays and because batteries and devices communicate – this is difficult to mimic if all one has is an external power meter that cannot cover the communication protocols of various battery manufacturers.[1] In contrast, this work uses the easy-to-access internal meter (a specialised battery fuel gauge chip with various compensations) for both model-building and in-vivo optimisation. Loosely related here are specialised profilers such as Trepn [20] for Qualcomm processors have their place, however, it is not trivial to integrate their results into the model-building process for arbitrary code involving entire smartphones. In contrast to these approaches, we use the battery's internal meter to obtain the energy readings for the modeling building. This Maxim MAX17050 fuel gauge chip compensates measurements for temperature, battery age and load [21] and it is an adequate substitute of an external meter if the measurement periods are sufficiently long [4].

In addition, the code instrumentation procedure used for modelling in this paper is simpler and less labour intensive than the techniques in [15, 16, 25]. For example, the *vlens* tool [25] calculates energy use of apps at source line level using an external meter readings combined with program analysis and statistical modelling. Another example is the *elens* tool [16], which is a technique based on program analysis and the Software Energy Environment Profiler (SEEP) that estimates energy usage of Android APIs. SEEP is a labour intensive and infeasible to maintain, as there are thousands of APIs in Android SDK, and they evolve rapidly at rate of 115 API updates per month [29]. In addition, the work in [11, 35] measures the energy use of instructions at microprocessor level. Their results show that the variation in energy use between different instructions is relatively small. We make this one of our assumptions, but validate the trade-off configurations in the end on the device nevertheless. Our approach is similar to the work of [13, 41], which uses a battery monitor unit to measure energy usage and correlates it with CPU utilisation. Our work also extends the modelling process to the relationship between energy and lines of code (LOC).

There are several studies that use SBSE to improve the energy efficiency of software on desktop platforms [7, 9, 34]. All of that work builds the models from external meter readings and uses a single-objective technique. In contrast to this, our research targets mobile devices, which have been shown to be a hostile environment for such experiments [3].

In terms of portable devices, [8] applied a multi-objective optimisation technique trading off energy consumption (measured externally) on a raspberry pi, and [27] utilised an optimisation approach to trade off energy use of an OLED screen (model-based) against a measure of user-experience. We utilise both generated models and live battery readings to discover energy-accuracy trade-offs both in-vivo and off-line.

3 METHODOLOGY

This section outlines the setup of the experiments described in this work. In the following we describe, in turn: the target application that we use to demonstrate our approach; the evolutionary search framework used; the fitness function; the methodology for defining the search space; and the initialisation process for the search.

3.1 Target Application

In this section, we first list the requirements that target applications must satisfy for a subsequent optimisation. Then, we introduce our chosen application, characterise its test cases and define how we measure the impact of optimisation on the application's behaviour.

To be considered a target for optimisation, we require open-source applications to satisfy the following requirements: (R1) widely used, for maximum impact; (R2) computationally intensive, for potential room for improvement; (R3) provide tests that allow for gradual deviations from target outputs.

Interestingly, many open source applications do not satisfy the last requirement, as tests tend to focus on functional property checks such as data extraction from files, listening to events, user interface tests, and so on.

Following a comprehensive search for applications that satisfy all requirements, we use Rebound[2] in this study. Rebound is a Java library that models spring dynamics. The spring models in Rebound can be used to create animations that feel natural by introducing real world physics to applications. For example, in complex components like pagers, toggles, and scrollers. Major apps that use Rebound include Evernote, Slingshot, LinkedIn, and Facebook Home.

The focus of our optimisation is Rebound's `Spring` class in the `com.facebook.rebound` package. This class implements a classical spring using Hooke's law with configurable friction and tension. Inside this class, the `advance` function is responsible for the physics simulation based on `SOLVER_TIMESTEP_SEC` sized chunks. The computations include, among others, Euler integrations and calculations of derivatives. Interestingly, some level of performance optimisation has already been performed, as evidenced by the source comment "The math is inlined inside the loop since it made a huge performance impact when there are several springs being advanced."

Rebound comes with 44 test cases that vary significantly in nature. Those that perform the actual physics calculations are most important for us: these are (i) relatively time consuming and (ii) deviations from the exact results may be acceptable if energy consumption is decreased as a result of a configuration change.

[1] In our preliminary testing with external meters, our modern devices would either not start or they would shut down abruptly when we would leave the battery communication pin(s) unconnected.

[2] Rebound Spring Animations for Android: http://facebook.github.io/rebound/, accessed 10 May 2018.

Interestingly, the original test cases do not result in a quality of 0, but in a tiny non-zero value. This is due to tests not resulting in exactly the spring speed and position values provided in the test oracle. To address this, we adjust the test oracles based on the actual output of the code on the device.

3.2 Evolutionary Framework

In our main experiments we use NSGA-III [12] to optimise two objectives: energy use by Rebound and the deviation of the output of the modified application from the original. We use this state-of-the-art algorithm to explore the multi-objective configuration spaces that result from our ways of measuring and predicting energy use. Individuals in the search space are variants of the Rebound application. These variants are created during search using deep parameter optimisation (DPO) [6, 40]. DPO is a genetic improvement technique [33] where variants are produced by mutating constants (deep parameters) found within its source code, and which are typically not accessible to a user. These deep parameters are exposed to the search process by automatically lifting them to be encoded as explicit constants.

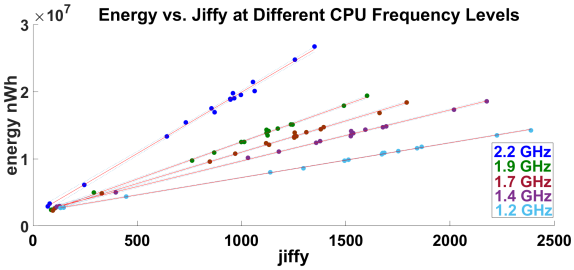
For Rebound, our framework exposes integer and double constants within the source code. This starts by replacing those constants with placeholders. The placeholders are calls to read each placeholder's value from a configuration file. In most genetic improvement research, modifications to the source-code require recompilation before evaluation. This can be costly – in our case, recompilation carries a penalty of 20-30 seconds. Encoding individuals as configuration files for the exposed parameters eliminates the cost of recompiling Rebound variants. The configuration file is read once per execution and thus incurs a fixed energy overhead though. As the file size remains stable, we assume this read overhead to be constant across any and all evaluations, and therefore it does not effect the outcomes of the search process.

3.3 Fitness Functions

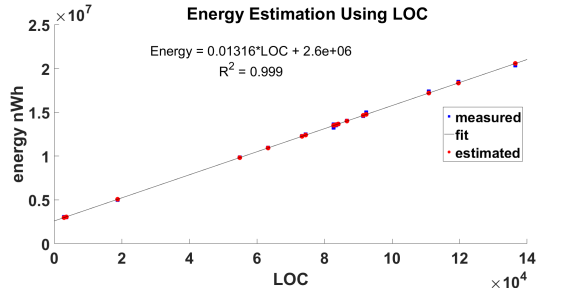
Two fitness functions are used in this work: the *energy* used by an individual program variant and the *accuracy* of that program variant.

3.3.1 Fitness: Energy. In our experiments, we compare three alternative methods to measure fitness: in-vivo measurement; on-phone measurement of CPU utilisation (Jiffies); and a Lines-of-code (LOC) proxy for energy use running on a computer. In all cases the proxies model CPU-usage.

In-Vivo Energy Measurement. One way to measure the energy use of a program variant is to perform experiments on a working platform – in-vivo – and sample an internal battery meter before and after the trial run. In our experiments our target platform is the HTC Nexus 9 running the Android 6 operating system. The special feature of this device is that it is equipped with the Maxim MAX17050 fuel gauge chip that compensates measurements for temperature, battery age and load [21], which provides an adequate substitute of an external meter if the measurement periods are sufficiently long [4]. Android 6 is used in our testbed as its market share of Android flavours was 32% and 25% for the second half of 2017 and first half of 2018, respectively[3]. In the remainder of this article, we will use nWh as the unit, as this is the battery gauge chip's provided resolution.



(a) Energy estimation using jiffies at different CPU levels.



(b) Measured energy vs. estimated using LOC at CPU frequency of 1.4 GHz.

Figure 1: Actual energy use of Rebound as a function of jiffies and LOC.

In our experiments we use a version of the methodology described in [3] to load a set of program variants to the phone via the Android Debugging Interface version 1.0.36 (ADB), and then cycle through the variants, sampling the internal meter before and after, and finally disconnecting ADB and charging the phone for the next generation. The internal meter is accessed through Android's BatteryManager. This API broadcasts these values with a frequency of 4Hz. The precision of this approach is significantly higher than ADB's own energy estimates [3], which are based on rough and uncompensated system models.

In running the optimisation process a great deal of care is required to avoid systematic noise induced by dynamic CPU speeds, effects of heat, non-linearity in battery response, overheads of memory logs, garbage collection, communication and UI devices, and sleep modes. For the detailed description, we refer the interested reader to [3].

Dealing with small energy signals. One problem specific to Rebound is test-harness overhead. As it is pointed out in [23], test duration is inversely proportional to smallest detectable impact. The easiest way to increase test duration is to repeat the whole test case several times to increase the energy signal. However, the test harness overhead for Rebound is bigger than the run-time of the software being optimised – by a factor of four-to-one. One can

[3] Statista – the Portal for Statistics: https://www.statista.com/, accessed 8 October 2018.

alleviate the overall overhead by repeating only the core application or function under test multiple times within each test case to ensure the test run spends a greater percentage of time running the program under mutation. Unfortunately, for Rebound this approach is not effective because the overhead in running each test case, e.g. state changes in the JVM caused by multiple runs and setting up listeners, is still large compared to the target code. In these cases, even with repetition, the code that is not subject to optimisation dominates the code being optimised. In addition, in memory constrained environments such as smart-phones, the impact of just re-running the tests fills up the application's allocated heap portion, which causes frequent Garbage Collector (GC) invocations to free the allocated memory. This notably increases the test time as well as it adds more noise to the collected energy signal [2, 18]. We address this problem in our experiments by instrumenting the code to be optimised with dummy loops after each line of code. This simple approach serves to amplify the effect of any change to the parameters of the original code. In a setting where code structure is being optimised this technique can be applied automatically with a tool such as JavaParser.[4]

Note that, by using dummy loops, we make the assumption that all lines of code are equal in terms of usage, however, different instructions can have small differences in energy consumption [35, 36]. However, in this setting, where the target code exclusively uses CPU and RAM, the assumption of uniform usage doesn't adversely affect search. Of course, final validation of the optimised configuration using the non-amplified code is required and this validation is presented later.

CPU Utilisation Model. To build the CPU utilisation model for the Nexus 9, the amplified version of Rebound, used for the *in-vivo* energy measurement above, is run with a set of 15 different configurations[5]. Each run is repeated eleven times at different CPU frequencies. During the run, the system statistics are accessed for system software clock expressed in *jiffy counts* as a measure of CPU utilisation. In these experiments, the precautions described in [3] were taken to minimise the effects of temperature, memory-consumption, file-system overhead, hardware and software governors, and other peripheral devices.

Figure 1(a) shows the mapping of jiffies to energy use, for different CPU frequency levels. As can be seen, the CPU utilisation expressed in jiffies and energy use is linear. While this finding for Rebound on our hardware aligns with other works in the literature [14, 39, 41], such relationships do not, by any means, hold in all settings and can even vary across devices [13, 28].

Lines-of-Code Model. As another basis for comparison, we use the number of executed lines-of-code (LOC), to estimate the energy consumption. In in this model we use experimental data from profiling the CPU at 1.4 GHz. It can be observed from Figure 1(b), the energy consumption linearly correlates with the executed LOC for Rebound. The R^2 of the model is 0.99, indicating a strong correlation. Moreover, the computed mean absolute percentage error is less than 1%.

[4]JavaParser, https://javaparser.org/, accessed on 10 May 2018.
[5]We use these configurations to vary the CPU workload.

3.3.2 Fitness: Accuracy. The second dimension of the fitness function is the accuracy of Rebound's output. When the Rebound library is run, it produces a single-dimensional trace of spring positions. In this work, the accuracy is determined by comparing the trace produced by the Rebound variant with the original Rebound. Variants whose traces closely track the original trace receive high accuracy. Major deviations from this trace receive low accuracy.

3.4 Refining the Search Space

The search space for the optimisation of Rebound are constants lifted from the code for the purpose of deep parameter optimisation. There are dozens of such parameters, which is an impractically large number to optimise with a quite limited number of function evaluations. To reduce the number of parameters, we conduct a sensitivity analysis to isolate the parameters to which the energy consumption of Rebound is most sensitive.

We profile Rebound by running its test suite and compute the code coverage to determine the frequently executed methods. In our case we find that most calculations are performed in just one Java class, `Spring`. For example, the previously mentioned `advance` method, which performs the physics calculations, is the second-most called method (9406 times).[6] These calculations are mainly executed inside a loop, which is considered an energy hotspot [1]. The most frequently called method is `isAtRest` (20340 times, also in `Spring`), which performs a rather simple calculation. All other methods consume relatively few computational resources. This class is therefore targeted exclusively as the other classes are unlikely to contain parameters that are worth optimising.

The class `Spring` contains 24 parameters. To check their impact on energy use, each parameter is multiplied by $(10)^x$, where x is an integer in the range $[-3, 3]$. The test is then executed 11 times for each parameter's setting. One test run takes about 15 seconds.

The parameters fall into three categories with respect to tests. First, *sensitive* parameters are those where the applied changes induce a significant change in energy use — these are worth optimising. Second, *insensitive* parameters create little change in energy consumption. Third, *too-sensitive* parameters cause a timeout in response to changes in parameter values. Within the category of *sensitive* parameters, alterations to values may reduce loop iterations [32], or disable certain costly branches [40] to reduce energy usage. Because we permit deviation from the test oracles, it is likely that trade-offs can be found to minimise the consumed energy at the expense of test quality [8]. For the purposes of optimisation, the sensitive parameters are represented by an n-tuple of numbers to form a solution. A fitness value for the objectives of energy and accuracy is assigned to each solution.

Among the 24 parameters in `Spring`, nine parameters can be classified as sensitive. Since the number of evaluations is limited in our experiments, we furthermore select from these nine only those parameters that reduce the overall CPU utilisation (i.e. CPU jiffies) by at least 20%. Table 1 shows the selected five parameters, the number of jiffies required to run the test suite and the reduction percentage. Interestingly, the impact on CPU use starts to appear only after at least two magnitudes of change in all parameters,

[6]Determined by Corbertura 2.1.1, available at http://cobertura.github.io/cobertura/, accessed 10 May 2018. The total class/line/branch coverage is 40%/61%/61%.

except for Spring_DOUBLE_24_1 which dramatically decreases the number of jiffies (reduction by 91%). These findings conform with previous research on deep parameter optimisation, which indicate that the majority of exposed parameters are not worth optimising [6, 40].

Parameter Name	Multiplication Factor	Jiffy	Reduction %
Original	n/a	1524	n/a
Spring_DOUBLE_26_1	1000	116	92%
Spring_DOUBLE_24_1	0.001	128	91%
Spring_DOUBLE_26_1	10	397	74%
Spring_DOUBLE_377_3	1000	1000	34%
Spring_DOUBLE_46_1	1000	1034	32%
Spring_DOUBLE_377_1	1000	1034	32%
Spring_DOUBLE_377_3	100	1048	31%
Spring_DOUBLE_377_1	100	1049	31%
Spring_DOUBLE_46_1	100	1181	23%

Table 1: The five selected parameters after the sensitivity analysis. The parameters' names reflect their properties. For example, Spring_DOUBLE_377_3 is the third floating-point number used in line 377 of the Spring class.

Figure 2 shows a comparison between the original configuration (default values) and the selected parameters for optimisation. As can be seen, the search space is non-monotonic. For example, changes to SPRING_DOUBLE_26_1 can drastically improve the energy efficiency after being multiplied by 10 and 1000, however, multiplying it by 100 (and by quite a few other numbers not listed here) results in timeouts.

3.5 Initialisation

While we could create the initial population (i.e., the initial set of program configurations) by generating random solutions, we attempt to maximise diversity by sampling both energy-hungry and energy-frugal values for parameter settings.

As with the sensitivity analysis, we base the seed population on the original program configuration, and then multiply selected parameters by factors of the form 10^x. The exponent here is randomly drawn from a Gaussian distribution with $\sigma = 3$, to allow us to cover an even greater space than the original sensitivity analysis.

Individuals are generated until the initial population is seeded with $\mu = 25$ valid parameter vectors. We limit the perturbations to only two parameters (dimensions) per solution, to reduce the number of timed-out (invalid) solutions generated: we found that one and two dimensional changes in one solution takes require than 50 trials while more changes take up to 120 trials. Figure 3 shows an example of an initial population.

4 EXPERIMENTS

As mentioned previously, we use NSGA-III [12], a genetic algorithm designed for multi-objective search, as implemented by the MOEA Framework.[7] Three experiments are conducted where energy use obtained by the internal meter, and via our jiffy and LOC models. The *in-vivo* experiments were conducted on Nexus 9 tablet running Android 6, whereas the off-line experiment (using the LOC model) was performed on a Windows 10 machine with 16GB memory and Intel i7-6700 CPU clocked at 3.4GHz.

Note that we use the algorithm purely for the purpose of navigating the trade-off space, and as part of this study to optimise small energy signals. For a current overview of current multi-objective algorithms, we refer the interested reader to [10, 24].

Any parameter is constrained to values between 0 and 10000; their original values are between 0 and 6. With a population size of $\mu = 25$, we seed the initial generation using the steps described above. With this setup we run for 1250 evaluations. Since two of the experiments run *in-vivo* (optimisation based on internal meter readings and the jiffy model), we use similar settings to those found in [3] to overcome Android's challenges for energy optimisation experiments. In addition, we limit the generation size to avoid any variation in voltage, as this is a real energy-based experiment. In case of having a generation full of invalid solutions (worst case scenario), the timeouts are responsible for longer runtimes, and thus for longer discharge phases, voltage drops, and less reliable readings.

4.1 Results

4.1.1 Configurations in the objective space. Figure 4 shows a summary of the results. In the left column, we show the evaluated solutions as well as the Pareto front obtained. In the right column, we focus on the nine to twelve solutions of the Pareto front and the values of the corresponding decision variables.[8]

Let us start with the topmost row, i.e., the results for the optimisation that ran in-vivo and that used the raw energy readings as provided by the battery sensor. As can be seen, there are nine non-dominated solutions. In terms of energy efficiency, the best solution found uses 2.4 mWh in the raw energy optimisation, and the optimiser took 991 fitness evaluations to find it. This compares favourably with 13.4 mWh used by the original configuration. On the other hand, its accuracy deviates by 1.2 on overall, and it passes only one test, making it the worst solution on the front in terms of accuracy.

Let us briefly investigate a particular solution to see the trade-off of how a configuration change affects energy consumption. In the run that used the raw readings, the solution with the second-lowest energy consumption (second marked red dot from the left) consumes 4.5 mWh and has an acceptable deviation from the test oracle. Although it fails to pass five test cases, its deviation is relatively small with average of average absolute deviation of 0.09. Figure 5 shows the results of testing the `Spring` positions/steps while being in motion on all six tests. As can be seen, after evolving the new values, the deviation is very small and might not even be noticeable by a user.

[7] MOEA Framework version 2.12 available at http://moeaframework.org, accessed 10 May 2018. We leave all variation operators and variation probabilities at their standard values.

[8] In the context of multi-objective optimisation, the optimal solutions are also referred to as non-dominated solutions, and they form the so-called Pareto front. In a minimisation problem, a solution x is considered non-dominated in comparison to another solution x^* when no objective value of x^* is less than x and at least one objective value of x^* is greater than x. For a more comprehensive introduction we refer the interested reader to [10, 24].

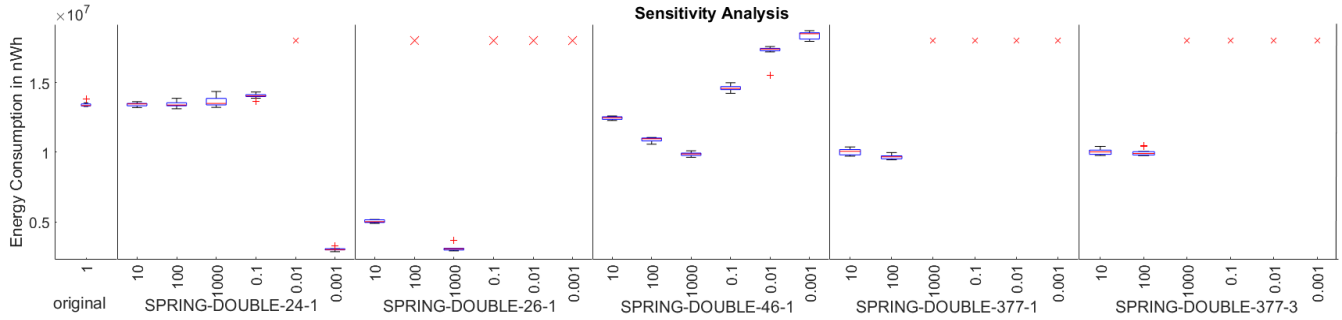


Figure 2: Comparison of the original configuration with modifications of the sensitive parameters. The numbers 0.1 to 1000 denote the factors by which the original value of the parameter is multiplied. Red crosses show timed-out configurations.

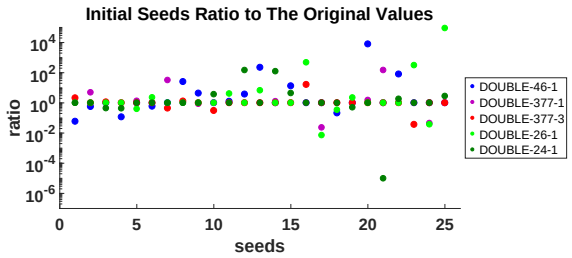


Figure 3: The amount of modification applied to each parameter in the seeded first generation of $\mu = 25$ solutions. Only two parameters are altered per solution (two dimensions).

Coming back to Figure 4, the second and third rows show the optimisation results obtained based on our jiffy and LOC-based energy models. In the extreme case, the energy use was reduced to 2.95 mWh and 3 mWh.

For a quick check of the diversity along the two objectives, we use the coefficient of variation (CV). For the optimisation based on raw readings, the CV values are 55% and 163% for energy use and accuracy. The CV values for the fronts resulting from the model-based optimisation are comparable: for energy/oracle deviation they are 56%/192% and 63%/186% for the jiffy and the LOC models.

4.1.2 Configurations in the decision space. The right column of Figure 4 illustrates the actual solutions of the Pareto fronts in the three experiments. To allow for an easy comparison with the original configuration, we provide the solutions as factors applied to the original configuration. These Pareto-optimal solutions are sorted from left to right in increasing order based on their increase in energy consumption (and thus in decreasing order of test deviation).

Quite surprisingly, the solutions of the three fronts are relatively similar. For example, DOUBLE_46_1 (blue) is almost always increased by about five orders of magnitude. This parameter represents the rest-speed threshold at which the `Spring` is determined to be at rest. Also, DOUBLE_24_1 (dark green) is modified by two to four orders of magnitude, and the others often just comparatively little. To us, this is additional evidence that (1) the developed models are consistent with the real world, and (2) the results are reproducible, even when using a different model or the real device.

When we have a closer look at the individual fronts, additional patterns emerge. For example, as energy consumption increases from left to right, DOUBLE_24_1 (dark green) appears to be decreasing in the model-based results. Regarding the other constants, higher order dependencies exist, so no one single parameter "drives" the tradeoff.

4.2 Pareto Front Validation

Next, we validate solutions found on the three Pareto fronts. The validation process consists of removing the dummy-for loops, and running the test suite by repeating the actual call to the `advance` function 1000 times. During the test run, the energy is measured by the internal meter. The test run is repeated 31 times for each solution.

Figure 6 shows the result of the validation and a comparison with running the default values of the parameters with the same settings. As can be seen, all of the optimised variants have a significant difference compared to the original settings. This indicates the feasibility of using energy models as a fitness function. Despite the battery's internal meter being less accurate than expensive external meters, the results demonstrate that it can be used for optimising energy efficiency *in-vivo*. This is because these types of internal meters are precise [4] and therefore can be used to rank solutions in terms of energy use. Also, this does not require developers to have a special skills to obtain energy readings.

To determine whether the difference between the amount of energy consumed by the original version and evolved variants of each subject is statistically significant, we use the right-tailed Wilcoxon rank-sum test, where the alternative hypothesis states that the median of the original configuration is greater than the median of the improved variant. We chose Mann-Whitney-Wilcoxon test due to the observation of having non-normal distribution. All obtained p-value for each test are less than 0.005, indicating a highly significant difference in the results. For instance, the p-values for the solutions *raw 1* and *jiffy 1* in comparison with the original configuration are $1.3 \cdot 10^{-4}$ and $1.4 \cdot 10^{-9}$, which we consider to be highly significant.

In order to better quantify our improvements, we measured the test framework's overhead by running the test suite with an empty advance function. It was found that the overhead amounts to 66%

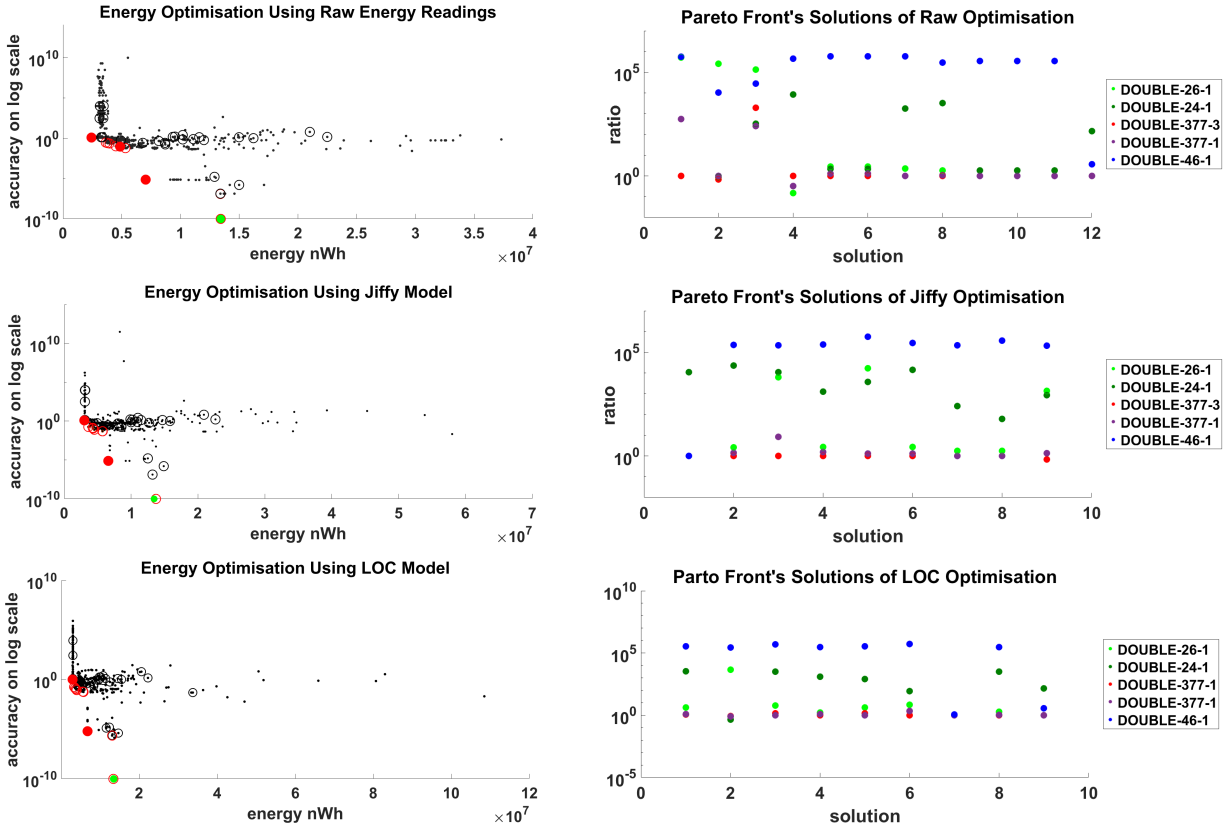


Figure 4: Optimisation results. Left: solutions in the objective space (black) with the initial population circled, the Pareto front circled red and highlighted solutions for the later validation marked as solid red circles; the original configuration in light green. Right: solutions on the Pareto front in the decision space.

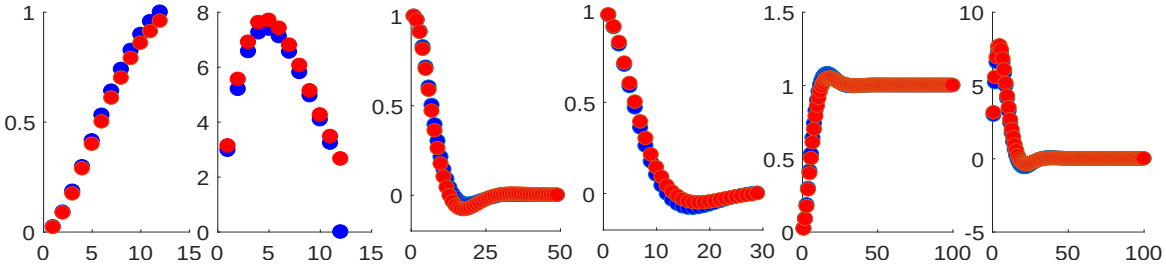


Figure 5: Spring's expected result (test oracle, red) vs. the actual result (blue) of the second marked solution from raw energy optimisation experiment on six test cases. Animation steps are on the x-axis and the spring's position or velocity is on the y-axis.

of the default configuration runs (based on the median of each set of the 31 runs of each setup). After deducting it from the results in Figure 6, the actual improvements in the energy efficiency of running Rebound's test cases using the found configurations range between 7-22% (based on the medians) across the seven shown solutions. We conjecture that Rebound's developers might not be aware of such an energy improvement at the expense of a slight deviation from the functional requirement.

Finally, we use a Nexus 6 phone running Android 6 to check if the evolved solutions can improve the energy efficiency on another device other than the Nexus 9. In terms of hardware specifications, both devices are drastically different. For example, The Nexus 6 is powered by a 2.7 GHz quad-core Snapdragon 805 processor with 3 GB of RAM, where as the Nexus 9 has 2 GB of memory and its system chip is NVIDIA Tegra K1 with a 2.3 GHz dual-core Denver CPU. It is worth mentioning that the battery fuel gauge on Nexus

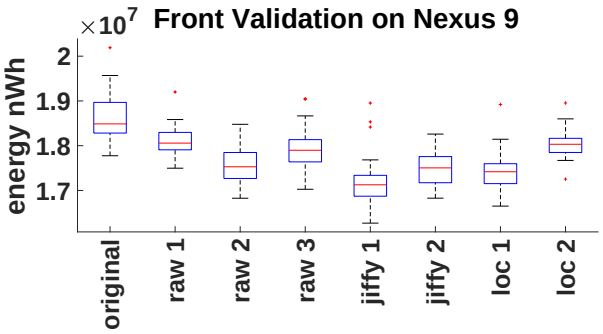


Figure 6: On-device validation of highlighted solutions from Figure 4; this is the non-amplified code. *raw 2* is shown in Figure 5. The median of the test overhead is 1.22nWh.

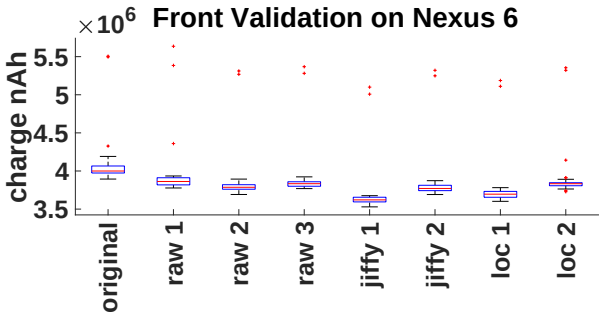


Figure 7: Validation of solutions from Figure 4 on Nexus 6.

6 reports the remaining charge in nAh, and we use it without any conversion.

Figure 7 shows the results of validating the marked solutions from 4 and the original configuration on Nexus 6 using the same settings mentioned earlier in this section. As can be seen, interestingly, the overall trend is similar to the results found on the optimisation platform (i.e. Nexus 9) though the two devices have different hardware specifications. In addition, the jiffy 1 variant still uses the lowest amount of charge among the validated solutions. Running the right-tailed Wilcoxon rank-sum test on them indicates the difference (to the original charge usage) is statistically significant. For example, the (p-value is $1.41 \cdot 10^{-9}$) when *raw 2* is compared to the original configuration.

5 CONCLUSIONS

The optimisation of non-functional properties of applications is of increasing interest: while developers generally lack the skill, search-based software engineering can assist with an automated approach. When it comes to minimising the consumption of energy, one has to deal with noisy sensors, huge search spaces, and long evaluation times.

In this article, we demonstrated that it is possible to detect small changes in energy consumption using code rewriting. This was required to explore the configuration space of an Android physics library. To speed up the optimisation process, we created models based on runtime and lines-of-code, which were sufficiently precise to guide the optimisation. The former still requires to be run on the device, however, it no longer requires to connect and disconnect the device for each configuration evaluation. The latter model can run entirely on the computer and thus is significantly faster.

The results show that substantial energy savings of up to 22% can be achieved for our target application (after deducting the test overhead), at comparatively little deviation from the functional requirement.

In the future, we will tackle video decoders that are embedded deep within the operating system, and the inherently noisy data communication.

6 ACKNOWLEDGEMENTS

Mahmoud Bokhari has been sponsored by Saudi Arabia Cultural Mission (SACM). Markus Wagner has been supported by ARC Discovery Early Career Researcher Award DE160100850.

REFERENCES

[1] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 588–598.

[2] S. Bhadra, A. Conrad, C. Hurkes, B. Kirklin, and G. M. Kapfhammer. 2009. An experimental study of methods for executing test suites in memory constrained environments. In *Workshop on Automation of Software Test*. 27–35.

[3] Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner. 2017. Deep Parameter Optimisation on Android Smartphones for Energy Minimisation: A Tale of Woe and a Proof-of-concept. In *Genetic and Evolutionary Computation Conference (GECCO) Companion (GI Workshop)*. ACM, 1501–1508.

[4] Mahmoud A. Bokhari, Yuanzhong Xia, Bo Zhou, Brad Alexander, and Markus Wagner. 2017. Validation of Internal Meters of Mobile Android Devices. *CoRR* abs/1701.07095 (2017).

[5] A. E. I. Brownlee, N. Burles, and J. Swan. 2017. Search-Based Energy Optimization of Some Ubiquitous Algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (2017), 188–201.

[6] Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. 2016. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV. In *Symposium on Search-Based Software Engineering (SSBSE)*. Springer, 238–243.

[7] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1327–1334.

[8] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr. 2018. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2827066 Accepted.

[9] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In *Search-Based Software Engineering*. Springer, 255–261.

[10] Shelvin Chand and Markus Wagner. 2015. Evolutionary many-objective optimization: A quick-start guide. *Surveys in Operations Research and Management Science* 20, 2 (2015), 35 – 42.

[11] Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. 2000. Cycle-accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI. In *International Symposium on Low Power Electronics and Design*. ACM, 185–190.

[12] K. Deb and H. Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evol. Computation* 18, 4 (2014), 577–601.

[13] Mian Dong and Lin Zhong. 2011. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In *Mobile Systems, Applications,*

and Services. ACM, 335–348.
[14] Jason Flinn and M. Satyanarayanan. 1999. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *2nd Workshop on Mobile Computer Systems and Applications*. IEEE, 2.
[15] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2012. Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In *1st International Workshop on Green and Sustainable Software*. IEEE Press, 1–7.
[16] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating Mobile Application Energy Consumption Using Program Analysis. In *2013 International Conference on Software Engineering*. IEEE Press, 92–101.
[17] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *11th Working Conference on Mining Software Repositories (MSR)*. ACM, 12–21.
[18] Ahmed Hussein, Mathias Payer, Antony Hosking, and Christopher A. Vick. 2015. Impact of GC Design on Power and Performance for Android. In *8th ACM International Systems and Storage Conference (SYSTOR)*. ACM, Article 13, 12 pages.
[19] Android Inc. 2017. Background locationOLD Limits. http://tiny.cc/locold
[20] Qualcomm Innovation Center Inc. 2014. Trepn Profiler. https://developer.qualcomm.com/software/trepn-power-profiler/tools Accessed 10 May 2018.
[21] Maxim Integrated. 2016. MAX17047/MAX17050 ModelGauge m3 Fuel Gauge. https://datasheets.maximintegrated.com/en/ds/MAX17047-MAX17050.pdf Accessed 10 May 2018.
[22] Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *4th International Workshop on Green and Sustainable Software*. IEEE Press, 8–14.
[23] William B. Langdon, Justyna Petke, and Bobby R. Bruce. 2016. Optimising Quantisation Noise in Energy Measurement. In *Parallel Problem Solving from Nature (PPSN)*. Springer, 249–259.
[24] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-Objective Evolutionary Algorithms: A Survey. *Comput. Surveys* 48, 1, Article 13 (2015), 35 pages.
[25] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *2013 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 78–89.
[26] Ding Li, Angelica Huyen Tran, and William GJ Halfond. 2014. Making web applications more energy efficient for OLED smartphones. In *36th International Conference on Software Engineering (ICSE)*. 527–538.
[27] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach. In *10th Foundations of Software Engineering*. ACM.
[28] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. 2011. Evaluating the Effectiveness of Model-based Power Characterization. In *USENIX Annual Technical Conference*. USENIX Association, 12–25.
[29] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 70–79.
[30] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89.
[31] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-grained Power Modeling for Smartphones Using System Call Tracing. In *European Conference on Computer Systems*. ACM.
[32] Fabio Pellacini. 2005. User-configurable automatic shader simplification. *ACM Transactions on Graphics* 24, 3 (2005), 445.
[33] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2017. Genetic Improvement of software: A comprehensive Survey. *IEEE Transactions on Evolutionary Computation* (2017). To Appear.
[34] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler Software Optimization for Reducing Energy. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 639–652.
[35] A. Sinha and A. P. Chandrakasan. 2001. JouleTrack-a Web based tool for software energy profiling. In *38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 220–225.
[36] Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. 2001. An accurate and fine grain instruction-level energy model supporting software optimizations. In *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 10.
[37] James Tiongson. 2015. Mobile App: Marketing Insights. http://tiny.cc/twg
[38] Narseo Vallina-Rodriguez, Pan Hui, Jon Crowcroft, and Andrew Rice. 2010. Exhausting battery statistics: understanding the energy demands on mobile handsets. In *Workshop on Networking, systems, and applications on mobile handhelds*. ACM, 9–14.
[39] Karel De Vogeleer, Gérard Memmi, Pierre Jouvelot, and Fabien Coelho. 2014. The Energy/Frequency Convexity Rule: Modeling and Experimental Validation on Mobile Devices. *CoRR* abs/1401.4655 (2014). arXiv:1401.4655
[40] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 1375–1382.
[41] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring. In *USENIX Conference on Annual Technical Conference*. USENIX Association.