Deep Parameter Optimisation on Android Smartphones for Energy Minimisation - A Tale of Woe and a Proof-of-Concept

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

Bobby R. Bruce
Centre for Research on Evolution, Search and Testing,
University College London, United Kingdom
r.bruce@cs.ucl.ac.uk

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

With power demands of mobile devices rising, it is becoming increasingly important to make mobile software applications more energy efficient. Unfortunately, mobile platforms are diverse and very complex which makes energy behaviours difficult to model. This complexity presents challenges to the effectiveness of off-line optimisation of mobile applications. In this paper, we demonstrate that it is possible to automatically optimise an application for energy on a mobile device by evaluating energy consumption *in-vivo*. In contrast to previous work, we use only the device's own internal meter. Our approach involves many technical challenges but represents a realistic path toward learning hardware specific energy models for program code features.

KEYWORDS

non-functional properties, mobile devices, multi-objective optimisation, dreaming smartphone, Android 6

ACM Reference format:
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 *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082519

1 INTRODUCTION

In recent years there has been growing interest in improving software systems' energy efficiency. This is partially due to environmental concerns (the majority of the world's electricity consumption is still derived from polluting fossil fuels [2]) but also usability issues present in battery-constrained mobile devices such as smartphones. The sale of smartphone devices has exceeded that of personal computers [11] with the average user spending 30 hours a month on mobile applications [32]. These applications consume the smartphone's limited energy reserves which can leave users with a drained battery at inopportune moments. A survey of mobile application complaints found that having resource intensive features has a larger negative impact on an application's rating than uninteresting content or a poorly designed interface [17].

It is know that mobile applications may be refactored to consume less energy and, therby, increase battery life. However, a study by Pang et al. [24] showned that developers typically lack the necessary knowledge to make software more energy efficient. While education may be a solution to this problem, a more cost-effective approach is to automatically refactor software to a more energy efficient state, entirely removing the need for developer intervention. Previous studies have shown that this is possible when given reasonable assumptions about an application's end-use such as the likely input data [8] or network usage [18].

Small improvements in energy efficiency are achievable without changing the functionality of a mobile application. Semantic preserving changes to design pattern implementation [22] and the resolution of energy bugs (instances when smartphones are unnecessarily left in high energy states) [5] have both been shown effective at reducing energy consumption. There are, however, limits to how much energy can be saved without sacrificing functionality [10]. Fortunately the majority (80%) of software engineers who work in energy-constrained systems are willing to sacrifice some requirements for reduced energy consumption [20]. Allowing a degradation in quality to fulfil non-functional requirements is part of an emerging field known as *approximate computing* [13]. Examples of approximate computing include Li et al.'s attempt to reduce the energy consumption of Android applications by decreasing the quality of visual interfaces [19] and Sitthi et al.'s work on reducing shader execution time by permitting faults in the graphics they produce [29].

In this paper we aim to reduce the energy consumption of *Rebound*, a Java physics library that models spring dynamics. It is used by popular Android applications like Evernote, Slingshot, LinkedIn and Facebook. During our optimisation, we allow for

GECCO '17 Companion, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3067695.3082519

an approximation of the intended output, with the goal of finding a set of configurations that represent trade-offs between energy consumption and faithfulness to physically correct animations.

This article is structured as follows. We introduce the target software of this study in Section 2, and describe our approach to optimising it by means of deep parameter optimisation in Section 3. We present our used target hardware in Section 4. Strongly related to this is our subsequent tale of woe in Section 5, which reports on some of rabbit-holes that we went down when turning a modern smartphone with a modern operating system (Android 6) into a testbed. A proof-of-concept evolutionary run is shown in Section 6, before we conclude this article with a summary in Section 7.

2 TARGET SOFTWARE: JAVA PHYSICS LIBRARY REBOUND

In this section, we first lay out our requirements that target applications need to satisfy for our 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.

Our requirements for open-source target applications are as follows:

- *R1*: widely used, for maximum impact;
- *R2*: computationally intensive, for potential room for improvement;
- *R3*: compilable in under a minute on a regular computer;
- *R4*: provide tests that allow for gradual deviations from the targets.

Interestingly, many open source applications do not satisfy the last requirement, as tests tend to focus on function property checks such as data extraction from files, listening to events, application of ciphers, user interface tests, and so on. Two noteworthy groups of applications are internet browsers and media players, however, they are not considered here as their compilation requires many resources and some of the media decoders are implemented in hardware, which means testing on different platforms is not straightforward.

Following a comprehensive search for applications that satisfy all requirements, we use Rebound[1] 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 target for our optimisation is the `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."

[1]Rebound Spring Animations for Android: http://facebook.github.io/rebound/, accessed 12 March 2017

Rebound comes with 44 test cases. These tests vary significantly in nature. For example, some tests check if the ID of a spring is set correctly, and if listeners work as intended. Most importantly for us are the tests that perform the actual physics calculations. 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.

In general, for a set of n tests with test oracles $T_1, \ldots, T_n$ and corresponding observed outputs $O_1, \ldots, O_n$, we measure the quality in three ways:

(1) *M1*: How many tests are passed, as determined by `assertEquals(`T_i`,`O_i`)`, $1 \leq i \leq n$?
(2) *M2*: If an array is to be produced, what is the average per-element deviation? Variations here can result in unrealistic looking animations.
(3) *M3*: If an array is to be produced, what is the average array length deviation? Variations here can result in too long/short animations.

Note that if for test i the output arrays of T_i and O_i differ in length, then *M2* considers only the first $\min(|T_i|, |O_i|)$ fields.

Interestingly, the original test cases do not result in a *M1* 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.

Under the above testing regime the original code has the following outcome:

(1) *M1*: all 44 tests are passed;
(2) *M2*: the average deviation from the values provided by the oracle is zero;
(3) *M3*: the average deviation from the oracle's array lengths is zero.

3 DEEP PARAMETER OPTIMISATION

Deep parameter optimisation [9, 33] is a genetic improvement technique [27] where the variation operations to be applied to a target application are done so by toggling deep parameters found within its source code.

What constitutes a 'deep parameter' is anything within code, not previously exposed to the user, which may exist in multiple known forms while preserving some fundamental functionality. For example, in Java, when a developer wishes to use a collection they must declare which subclass of the abstract superclass `java.util.Collection` to implement (e.g. `java.util.ArrayList`, `java.util.HashSet`, etc.). Each of these subclasses consume different amounts of energy depending on how they are used [21, 26]. In the vast majority of cases the developer will choose an implementation based on his own intuition or preferences; utilising little information on how this may effect the software system's performance. In essence, the developer hard-codes these parameters. In deep parameter optimisation we expose these parameters to be toggled and then optimised using a search-based approach [15].

Within this investigation (as in previous investigations of using deep parameter optimisation to reduce energy consumption [9]) we expose integer and double constants. To do so we start by replacing

integer or double constants with placeholders. These placeholders are calls to read that 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 – within this investigation, recompilation carries a penalty of 20-30 seconds. With this setup the configuration file, which we may conceptualise as the exposed parameters, can be modified any number of times without recompilation. The configuration file is read once per execution and thus incurs a fixed energy overhead though, since this is constant across any and all evaluations this does not effect the impact of our results.

While one could begin tuning the parameters at this stage, it would be inefficient. Previous research on deep parameter optimisation has shown that the majority of exposed parameters are not worth optimising [9, 33]. We categoise deep parameters as falling within three categoties – those that too insensitive (large changes have no effect on the target property/properties), those too sensitive (small changes break hard constraints), and those worth optimising. In this investigation we start by profiling our target application, Rebound, and selecting only those files which consume large amounts of energy for optimisation. 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 methods (9406 times).[2] The most frequently called method is `isAtRest` (20340 times, also in `Spring`), which performs a rather simple check. All other methods are computationally uninteresting, and only called a few dozen times, if at all. We therefore choose to target this class exclusively as the remainder are unlikely to contain parameters that are worth optimising (i.e. they are too insensitive).

Before exposing the parameters within `Spring` we replace all instances of `{variable}++` with `{variable}+=1` and all instances of `{variable}--` with `{variable}-=1` as this improves the search-space by giving us more parameters to target for optimisation. Once this is done we expose 38 parameters from `Spring`. We then, for each parameter, increment its value by 1 if it is an integer or by 10% if a double (all doubles are non-zero). If this results in the program crashing when run we tag this parameter as unmodifiable as it is too sensative. Otherwise we multiply its value by 10 (after the incrementation). If this results in a program in which energy consumption exists within the 95% confidence interval of the unmodified application's energy consumption (determined by running the Rebound 100 times, measuring its energy each time) we tag the parameter as unmodifiable as it is too insensative. After this has been done for each parameter we are left with a set of parameters which we have not tagged as unmodifiable at any stage. These are the target parameters for deep parameter optimisation. In the case of `Spring`, we are left with 19 of the original 38 parameters.

With these parameters we may toggle them, thereby altering the software. These alterations may reduce loop iterations [25], or disable certain costly branches [33] to reduce energy consumption. Given we permit output approximation, it is likely trade-offs can be found to reduce energy consumption at the expense of output quality [10]. Though, at this abstract level, the problem representation is simply an n-tuple of numbers which may be assigned a fitness value for each objective producible for any given variant. Given this, it is an ideal candidate for optimisation using a genetic algorithm [30]. As we wish to optimise for multiple objectives (see Section 2) we utilise NSGA-II [12], a genetic algorithm designed for multi-objective search, as implemented by the MOEA Framework.[3]

We limit any parameter to have a minimum value of 0 and a maximum value of 64 (the parameters, when exposed, have initial values between 0 and 6). With a population size of $\mu = 20$, we seed the initial generation with the original solution (i.e. the parameters as exposed from the initial, unmodified application) and those close to the initial solution in the search space by iterating through the parameters and generating variants equal to the original but with one variable being incremented by 1 in order to introduce some initial diversity around the official parameter choice.

4 TARGET HARDWARE: ANDROID SMARTPHONES

4.1 Hardware Platform

Modern mobile phones are equipped with battery fuel gauge chips that report the voltage, current and remaining energy within the battery [3]. The target devices for our experiments are the HTC Nexus 9 and the Motorola Nexus 6. Both are equipped with the Maxim MAX17050 fuel gauge chip that compensates measurements for temperature, battery age and load [1].

For the optimisation of energy consumption, we solely rely on the energy readings as provided by the battery chip. This is in contrast to some existing work (e.g., [16]) which relies on external meters. A brief practical characterisation of the internal battery meters on the Nexus 6 and Nexus 9 is included in [7]. There, the authors outline results for validating the precision of the internal meters under various workloads. The internal meters are deemed sufficiently precise if the experiments are long. Also, based on our experience with external meters, their setup can come with unexpected electronic challenges. For example, we observed voltage drops and system crashes due to cheap alligator clips and corroding copper strips.

4.2 Software Framework

Our software framework includes a data logger, hardware component controller and battery monitor. The data logger samples hardware settings and utilisation data such as CPU frequency and load, screen brightness and network traffic. The controller's main job is to create test scenarios. It activates, deactivates and applies workloads on hardware components. For example, while profiling the screen, it changes and fixes its brightness, as well as it turns off other components and fixes the CPU frequency.

The battery monitor records the power consumption data such as the remaining energy, voltage and drawn current during each test session. Accessing the battery chip's values can be done through the battery API, such as Android's BatteryManager class. This API broadcasts these values with a frequency of 4Hz.

[2]Determined by Corbertura 2.1.1, available at http://cobertura.github.io/cobertura/ ,accessed23March2017.. The total class/line/branch coverage is 40%/61%/61%.

[3]MOEA Framework version 2.9 available at http://moeaframework.org, accessed 23 March 2017. We leave all variation operators and variation probabilities at their standard values.

5 GETTING THE EXPERIMENTAL SETUP RIGHT - A TALE OF WOE

Here, we ~~lament~~ report the various encountered challenges. The discussion is divided into expected challenges and unexpected challenges. We report on this because genetic improvement of energy usage is only as reliable as the measurements [14].

5.1 Expected Challenges

5.1.1 System Behaviour. It is important to note that both considered devices are complex with many communication interfaces, controller chips, and multiple CPU cores, where much of the device behaviour is controlled by the operating system. The operating system, Android 6.0.1, is itself is complex, with system and user processes running in parallel with elaborate power consumption management in place.

To minimise the noise from these complex systems and to maximise the relative strength of the energy signal from our experiments it is important to reduce the energy footprint produced by background processes. To this end we deactivated communication interfaces using the flight mode. This prevents processes from transmitting data, which has proven to substantially impact energy consumption, even when occurring in short bursts [6].

We also put the display in sleep mode, which reduces the power drawn for the display light and GPU. Turning the screen off allows the system to enter the so-called Doze-mode [4], which was introduced in Android 6, and which deactivates a number of system services that would otherwise inject noise into the energy signature of the experiments.

Another potential source of noise is the system dynamically adjusting CPU speed according the current workload. To avoid this issue we fix the speed of all cores to the same value.

5.1.2 Sampling-Frequency-Induced Error. Some sampling error is induced by the fact that the battery fuel gauge can be sampled with a maximum frequency of 250ms. This means that some noise is introduced by gaps between the start and finishing time of the measured process and the time the battery is sampled. We minimise the impact of this low sampling frequency by running each individual 20 consecutive times and recording the fuel-gauge samples only before and after these runs. The use of 20 consecutive runs also serves to increase the measured magnitude of energy use which further reduces the impact of random noise in the fuel-gauge. Finally the large number of runs also smooths out random variations in runs caused by sporadic drains on energy caused by system processes.

Note that the expected causes of measurement error above can be accounted for by reasonably standard approaches to sampling and controlling the run-time environment. Next we describe some unexpected challenges which require interventions which are specific to this domain.

5.2 Unexpected Challenges

5.2.1 System Behaviour. There are a number of unanticipated challenges presented by system behaviour. One unexpected interaction stemming from having the display in sleep mode is that the system will go into Doze-mode after the experiment starts. Doze-mode impacts the experiment by suspending and rescheduling background processes including those we use to log data and run the test suite of Rebound library. As a consequence, the test execution time increases drastically from seconds to hours in some cases. While the existence of Doze-Mode has benefits, it can be problematic in settings such as these experiments. To counter this problem we use partial wakelocks which prevents the system from suspending our processes. In addition, temporarily activating the display after each generation of the evolutionary process allows our framework to run Rebound's test cases normally. Needless to say, while the screen is on, the test execution is suspended.

5.2.2 Android Debug Bridge. Android Debug Bridge (adb) is a command-line tool by which developers can communicate with Android devices. It supports a variety of actions such as copying files to and from the device and installing/un-installing applications. adb consists of a client to initiate commands from the development machine, a daemon to run execute command on the Android device, and a server to manage the communication between the client and the daemon.

In our framework, adb is used to install code on target devices. In early experiments the Rebound library was compiled, transferred to, and installed on the device for every change in its parameters. Unfortunately, these operations took up to a minute to complete, making iterative search impractically slow. Moreover, app transfer and installation could fail due to adb server instability. Failure modes included, the connected device going off-line and interference with the communication port by other Android services. To reduce deployment time the Rebound library was modified to read its code parameters from a configuration file - thus avoiding the need to compile and transfer the application. To address the adb connectivity issue a programmable USB hub was configured to automatically drop and restore the link to the device whenever the device goes offline. The framework is also configured to poll for devices using `adb devices` and to restart the adb server in the case of interference on the communication port.

5.2.3 Temperature. Temperature variations during experimental runs produced an unexpected source of systematic noise in our experiments. In preliminary testing we observed that the battery temperature increased after many successive runs of Rebound. This temperature increase was observed to increase the fuel consumption of the same program when run repeatedly over time. Figure 1 shows how the rise of battery temperature correlates with increasing energy consumption on the same program run 1000 consecutive times (100 trials of ten Rebound runs each) on the Nexus 6 device. This trend of increasing energy consumption for the same code over time is extremely problematic in the context of an evolutionary run using NSGA-II. This is because the very first variant of the program will be on the Pareto frontier in the dimension of energy consumption. If the energy consumption increases as a function of temperature, and the temperature increases as a function of the number of evaluations, then it becomes progressively more difficult for genuinely improved individuals to dominate the starting program variant.

To see if it was possible to learn the relationship between temperature and power consumption – and therefore compensate for it – we collected energy consumption for the same Rebound benchmark

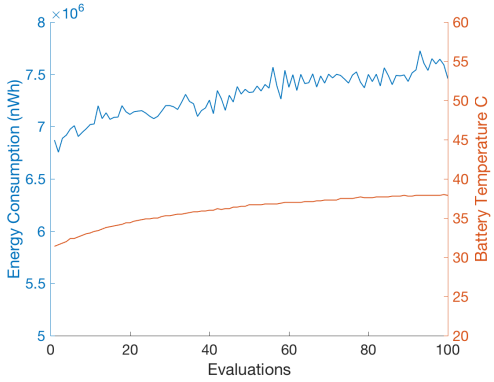

Figure 1: Battery temperature (red) and average energy consumption (blue) over 100 consecutive trials of 10 runs each on the same benchmark.

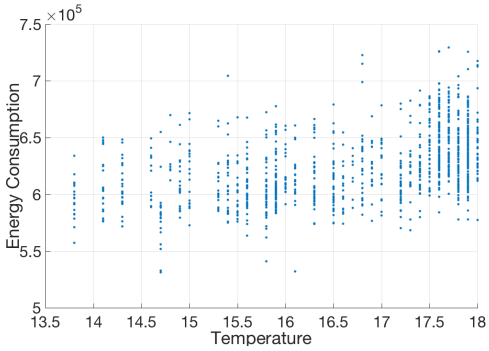

Figure 2: Scatter plot of temperature vs. energy consumption on the same benchmark.

in varying temperatures.[4]. Figure 2 shows a scatter plot relating temperature to energy readings from the same benchmark as previous. As can be seen, there is a general upward trend in the data.

We then ran non-linear regression on the resulting data using the GPTIPS2 [28] symbolic regression package. The learned function is: $e = 357t^2 - 6180.0t + 608000$ where e is the energy consumption of the benchmark in nWh and t is temperature in degrees Celsius. The actual-vs-predicted curves for the sorted temperature data is given in Fig 3. Note, that the data is sorted by temperature from left-to-right. Observe that there is large variance of energy consumption at each temperature level. Moreover, the variance seems to get larger as temperatures rise. This noisy relationship is borne out if we sort actual versus predicted energy consumption by error level, as shown in Figure 4. The figure shows that the distribution of energy consumption is bi-modal and the shape of the predictor function appears to be influenced by this. These systematic variances seem to indicate that there may be two populations of energy sensor readings from the Nexus 6.

To avoid this problem we switched to the Nexus 9 – which exhibited less systematic variation in energy readings. However, given,

[4]The experimental rig was placed in the refrigerator to extract some cooler readings.

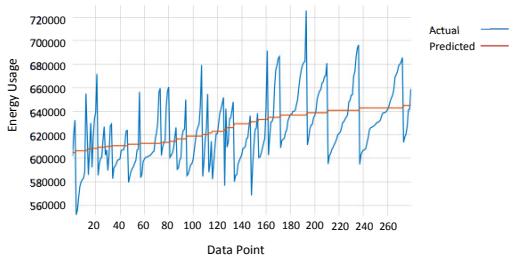

Figure 3: Actual energy consumption (blue) vs. predicted energy consumption (red) as a function of temperature.

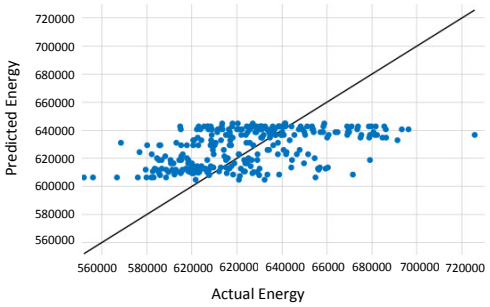

Figure 4: Fit of the regressed temperature function. As can be seen, the errors are bi-modal and the fitting function appears to be influenced by this.

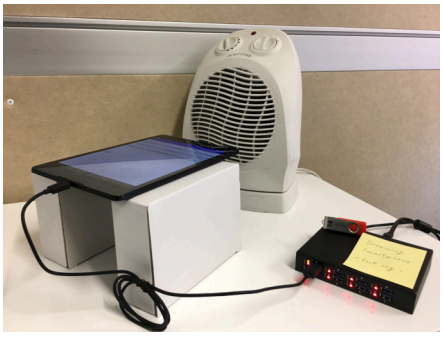

Figure 5: Test rig to limit variations in temperature in the Nexus 9 platform.

the difficulty in determining an accurate relationship between temperature and energy consumption, we decided instead to setup a test rig to control temperature by using a desk-fan coupled with sufficient egress for airflow around the device. This new rig is pictured in Figure 5.[5]

5.2.4 Processor Throttling. The test rig described above moderated rises in temperature but not enough to prevent a hardware

[5]For a video demonstrating an earlier setup, see https://www.youtube.com/watch?v=xeeFz2GLFdU

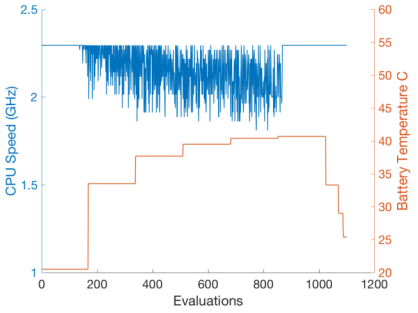

Figure 6: Battery temperature vs. processor speed over consecutive runs. As battery temperature rises the processor speed is throttled.

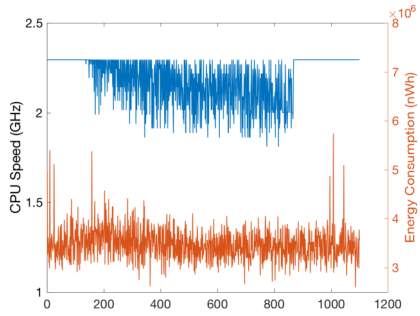

Figure 7: Processor-speed (blue) vs. energy consumption (red) during an evolutionary run.

CPU governor in the device from triggering reductions in processor speed. The effect of this fail-safe is shown in Figure 6. The figure seems to indicate that throttling is activated by steep rises in temperature rather than high absolute values in temperature. The throttling also seems to have the desired protective effect in slowing rises in temperature. This throttling has the potential to impact on the energy consumption of benchmarks. Figure 7 traces processor speed and energy consumption. The lower CPU speed leads to longer execution times on the Rebound benchmark but appears to lead to slightly less energy consumption. There also seems to be less variation in execution time at lower processor speeds.

Informed by these results, we throttled processor speeds using the system governor. We found a speed of 1.428GHz allows the benchmark to run in a tolerable time-frame whilst reducing measurement variability and temperature increases.

5.2.5 Log Files and Memory. Further experiments with reduced processor speed revealed that per-run energy consumption still increased over multiple runs – albeit at a slower rate than before. Preliminary investigations revealed that the increased energy consumption seemed to correlate with the size of logs and the size of the memory footprint of the Java test-harness used to run each generation. We re-designed the logging procedures to reduced the memory footprint of the generational log to 500kB. This reduction

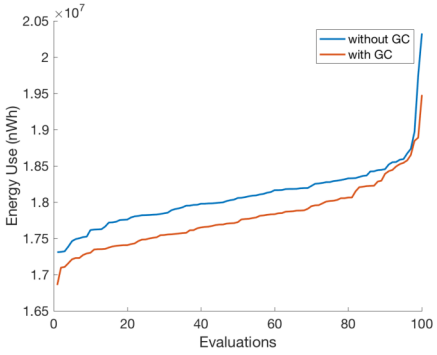

Figure 8: Sorted energy-use data for 100 runs with calls to the GC (red) and without calls to the GC (blue).

removed some of the variability energy consumption but an increase in energy consumption over multiple runs persisted. For further improvement we attempted to reduce the memory footprint of the test harness by calling the Java garbage collector (GC) every 250ms. This reduced overall energy consumption level and growth. As an illustration of the impact of calling the GC Figure 8 shows the sorted energy consumption data for a sequence of 100 runs with (red) and without (blue) garbage collection. After checking that both the with-GC and without-GC data were normally distributed (with a one-sample Kolmogorov-Smirnov test) we applied a t-test and confirmed the with-GC runs were significantly less than the without GC runs with $p \ll 0.001$. After further experiments it was determined that the same effect could be achieved by calling GC after each generation.

Informed by explorations described above, the experiments described in the next section were run with a processor speed 1.428GHz, using the physical setup shown in Figure 5.

6 PROOF-OF-CONCEPT

In the following, we report on our actual experiments performing deep parameter optimisation of a Java physics library as described in the previous sections.

6.1 Evaluation times

Due to the communication and framework overheads, our evaluations are relatively time-consuming. adb reports 20 Rebound runs have a total runtime of approximately 30 seconds. However, the initialisation of the individual runs and eventual clean-up by the framework results in a total time of approximately 50 seconds per effective evaluation of a Rebound configuration on the device. Also, in order to allow for variation in Rebound's runtime, and to account for variations in logfile write times, sleeping threads, and other Android peculiarities, we set the time-out per configuration evaluation to two minutes.

As running tests consumes energy, we recharge the device after each generation. Interestingly, different USB cables result in different charging currents and thus in different charging times needed. For example, one cable's charging current is approximately 0.8A (as reported by the USB hub's software), whereas a "better" cable allows

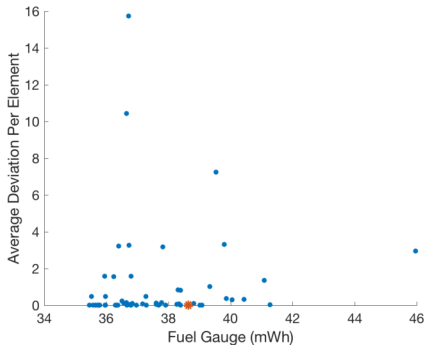

Figure 9: Scatter plot of energy vs. accuracy. Shown are the evolved configurations (blue) and the original one (orange).

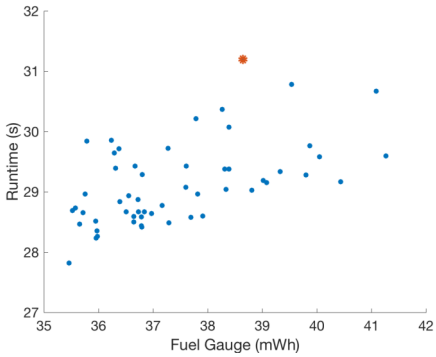

Figure 10: Scatter plot of energy vs. accuracy. Shown are the evolved configurations (blue) and the original one (orange). One configuration at about (46mWh,40s) is not shown.

for 1.4A. Based on preliminary experiments, we have conservatively chosen a charging time of 20 minutes between generations.

The above means that a single generation of $\mu = 20$ solutions takes approximately 1 hour on the device. While this seems costly, it is a big jump over some existing work, where a single configuration is evaluated by measuring the time needed for the device to run dry from 100% charge to 0% charge – which typically takes hours.

Note that, as a welcome side-effect of our recharging strategy between generations, we can keep the battery close to its maximum charge. Within each generation, we observed only very minor drops in the battery voltage, for example, from 4,214V to 4,201V over a duration of approximately 15 minutes. This greatly reduces potential measurement drift by the fuel gauge chip.

6.2 Results

In the following, we report on the results of a successful experiment, which ran for 17 generations. The first objective was the minimisation of energy, and the second objective was the minimisation of deviation from the oracle's values. We also recorded the array length deviations *M3*, however, we only observed two cases: either the length deviation was zero, or the entire test timed out.

After removing duplicates and timed-out solutions, 55 evolved Rebound configurations remain that are different from the original one. Figures 9 show these in the objective space, with the original configuration highlighted. Significantly, we can see solutions that consumed less energy at the cost of an decreased accuracy. In total, the 56 solutions achieve 43 different levels of accuracy. Their overall distribution of energy consumption covers quite a range, with the overall average being 37.5mWh and a standard deviation of 1.9mWh.

In terms of code features, it is difficult to gain consistent insights from the produced configurations, as we have 19 decision variables, 44 test cases with physics simulations, and noise in energy and time measurements. Out of the 19 variables, nine have direct influence on the mathematics involved in the spring simulation. When these are changed, then the resulting calculations deviate from the oracle. Additional experiments are necessary to further reduce the noise in the energy and time measurements to better understand the influence of certain parameters on the algorithms behind the simulations and thus on the observed energy consumption.

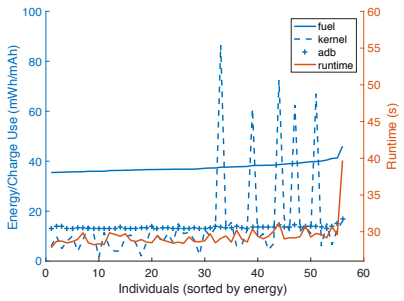

Figure 11: Correlation of energy with charge measurements and runtime. *fuel* is measured in mWh (correlation with runtime $r = 0.7382$), *kernel* and *adb* is charge in mAh ($r = 0.1479$, $r = 0.7727$).

Given the amount of technical difficulties described here, one might wonder if we could use a proxy function instead on mobile devices that do not have a dedicated battery chip installed. As one would expect, energy consumption and runtime (as reported by adb) are correlated in our experiment, as we can see in Figure 10. However, this is just a moderate, positive correlation with a correlation coefficient $r = 0.7382$. Other data sources that might be able to serve as proxies for energy consumption are Android's kernel-based estimations (which are based around imprecise and static power profiles for the entire device) and adb's own reports (which are also based on power profiles while considering time slices allocated to tasks).[6] We show the measurements for the 56 configurations in Figure 11. While the kernel's estimates are better ignored, adb's correlation with runtime is comparable to the one of the battery chip. Interestingly, the measurements of adb and the chip are not very highly correlated ($r = 0.758$), but this might again simply be the consequence of noise in the measurements. A limiting factor of adb's estimates when used for short experiments is its low resolution, which discretises the objective space unnecessarily[7] and thus can pose challenges for optimisation algorithms.

[6] The resolution of the kernel's and adb's estimates are 0.5mAh and 0.1mAh.
[7] adb/kernel: 14/33 different charge estimation values for the 56 configurations.

7 CONCLUSIONS

As shown in this article, the evolution of software configurations with modern mobile devices in-the-loop is feasible. However, our path to achieving this was littered with smaller and larger stumbling blocks, with some visible from afar and some only upon close scrutiny of results.

Our created test platform is quiet enough to allow us to see signals stemming from CPU utilisation. In the future, this will enable us to automatically learn energy consumption models for code segments. This will become increasingly important for energy-aware applications, as the operating system keeps evolving. For example, Android 8 will have a new and more restrictive approach to running apps in the background by throttling access to services [31]. While promising better battery life by means of operating system control, this will also immediately invalidate some existing consumption models for applications.

While there is a good chance for some of the future consumption models to be transferable across a range of mobile device models, we expect a degree of device-dependency of the models due to different CPU architectures in the over 24,000 different Android phones available [23]. This means that discovering the models will have to be on a per-platform basis. If we may express a wish, then we would love for smartphone manufacturers to spend an extra $1 on an accurate internal battery chip.[8]

ACKNOWLEDGMENTS

We thank our team members Younis Altoma, Yuanzhong Xia, and Bo Zhou for their support in various phases of this project. This work has been supported by the ARC Discovery Early Career Researcher Award DE160100850.

REFERENCES

[1] MAX17047/MAX17050 ModelGauge m3 Fuel Gauge. URL https://datasheets.maximintegrated.com/en/ds/MAX17047-MAX17050.pdf. Accessed November 2016.

[2] International Energy Agency. Key world energy statistics 2015, 2015. URL http://www.iea.org/publications/freepublications/publication/key-world-energy-statistics-2015.html.

[3] *Android Power Profiles.* Android. URL https://source.android.com/devices/tech/power.html. retrieved 03/2016.

[4] Android Developers. Optimizing for doze and app standby. URL https://developer.android.com/training/monitoring-device-state/doze-standby.html. Accessed 23 March 2017.

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

[6] Mahmoud Bokhari and Markus Wagner. Optimising energy consumption heuristically on android mobile phones. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) Companion*, pages 1139–1140. ACM, 2016.

[7] Mahmoud A. Bokhari, Yuanzhong Xia, Bo Zhou, Brad Alexander, and Markus Wagner. Validation of internal meters of mobile android devices. *CoRR*, abs/1701.07095, 2017. URL http://arxiv.org/abs/1701.07095.

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

[9] Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. Deep parameter optimisation for face detection using the viola-jones algorithm in OpenCV. In *Proceedings of the Symposium on Search-Based Software Engineering (SSBSE)*, pages 238–243. Springer, 2016.

[10] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. Approximate oracles and synergy in software energy search spaces. Technical report, Research Note RN/17/01, Department of Computer Science, University College London, 2017. URL http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/RN_17_01.PDF.

[11] Canalys. Smart phones overtake client PCs in 2011, 2012. URL https://www.canalys.com/static/press_release/2012/canalys-press-release-030212-smart-phones-overtake-client-pcs-2011_0.pdf. Accessed on 24 March 2017.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[13] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 2013 European Test Symposium (ETS)*, pages 1–6. IEEE, 2013.

[14] Saemundur O. Haraldsson and John R. Woodward. Genetic improvement of energy usage is only as reliable as the measurements are accurate. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 821–822. ACM, 2015.

[15] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[16] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 12–21. ACM, 2014.

[17] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[18] Ding Li and William GJ Halfond. Optimizing energy of http requests in android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 25–28, 2015.

[19] Ding Li, Angelica Huyen Tran, and William GJ Halfond. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 527–538, 2014.

[20] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE), pages=237–248, year=2016.*

[21] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th Int. Conference on Software Engineering (ICSE)*, pages 503–514. ACM, 2014.

[22] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 623–626. IEEE Press, 2015.

[23] Open Signal. Android fragmentation visualized, 2015. URL https://opensignal.com/reports/2015/08/android-fragmentation/. Accessed on 24 March 2017.

[24] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.

[25] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics*, 24(3):445, 2005.

[26] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS)*. ACM, 2016.

[27] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic Improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[28] Dominic P Searson, David E Leahy, and Mark J Willis. Gptips: an open source genetic programming toolbox for multigene symbolic regression. In *Proceedings of the International multiconference of engineers and computer scientists*, volume 1, pages 77–80. Citeseer, 2010.

[29] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic Programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152, 2011.

[30] Mandavilli Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[31] The Guardian. Google unveils android o, promising better battery life. URL https://www.theguardian.com/technology/2017/mar/22/google-unveils-android-o-promising-better-battery-life. Accessed 23 March 2017.

[32] James Tiongson. Mobile app: Marketing insights, 2015. URL https://www.thinkwithgoogle.com/articles/mobile-app-marketing-insights.html. Accessed on 24 March 2017.

[33] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1375–1382. ACM, 2015.

[8] Price per MAXIM17050 was USD 1.51 in 2013, see http://tinyurl.com/maxim17050press2013, accessed on 24 March 2017.