2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 978-1-6654-9596-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-NIER55298.2022.9793522

A Black Box Technique to Reduce Energy Consumption of Android Apps

Abdul Ali Bangash, Karim Ali, Abram Hindle
{bangash,karim.ali,abram.hindle}@ualberta.ca
Department of Computing Science, University of Alberta
Edmonton, AB, Canada

ABSTRACT

Android byte-code transformations are used to optimize applications (apps) in terms of run-time performance and size. But do they affect the energy consumption during this process? If they do, can we employ them to reduce an app's energy consumption? Given that most existing energy optimization techniques require developers to modify their code, a byte-code level modification technique will save developers' time and effort. In this paper, we investigate if byte-code transformations combined with genetic search can reduce an app's energy consumption. After applying our technique on four real-world apps, we find that some combinations of the byte-code transformations reduce the energy consumption by up to 11%.

ACM Reference Format:
Abdul Ali Bangash, Karim Ali, Abram Hindle. 2022. A Black Box Technique to Reduce Energy Consumption of Android Apps. In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, Pittsburgh, PA, USA, 5 pages. https://doi.org/10.1145/3510455.3512795

1 INTRODUCTION

Smartphone apps have become an essential part of our daily lives. From online education and shopping, to work and business meetings, many essential tasks are performed using apps. Given the limited energy stored within smartphone batteries, and that apps consume energy [35], developers aim to develop apps that provide a rich set of features that are also energy efficient [31, 36]. Developers are also concerned because battery consuming apps are left with poor user reviews leading to decreased marketshare [24, 35].

Prior work have shown that apps' energy consumption can be reduced by adopting energy-efficient design patterns [13], APIs [6, 18, 30], and UI elements [29]. Unfortunately, most of these approaches require developers to make source-code level changes that leave them with no choice but to: (1) change their code structure [3, 14] that could affect quality attributes such as maintainability, (2) select a specific API [6, 18, 30, 32] that could affect developers' options, or (3) change the user-interface (UI) structure [28, 41] that could compromise the design aesthetics of graphic designers. Instead, we propose a byte-code level black-box approach towards reducing the energy consumption of Android apps.

In this paper, we investigate if byte-code level transformations may reduce the energy consumption of Android apps. This black-box approach abstracts away energy consumption related details from the developers, which developers are usually unaware of [34, 35], and let them focus on the functional aspects of their app.

To evaluate the effect of byte-code transformations on Android apps' energy consumption, we use byte-code transformations offered by the Redex framework [22], a framework, designed by Facebook, to reduce app size and improve run-time performance. We employ genetic search to search for combinations of transformations that would reduce energy consumption the most, out of the many transformations available in Redex. Our algorithm first takes an app as input and generates multiple transformed apps by applying alternative combinations of byte-code transformations. We then measure the energy consumption of each transformed app to find the combination of transformations that transformed the app to consume least amount of energy.

Our results show that byte-code transformations, when applied on real-world apps, along with genetic search, reduces energy consumption by up to 11%.

2 PRELIMINARY STUDY

Before implementing our approach, we conducted a preliminary study to find out if byte-code transformations affect the energy consumption of Android Apps. To measure that effect, we first chose a set of transformations from the ones offered by Redex [22]. Table 1 provides a brief description of Redex's transformations. For the preliminary study, we randomly choose three transformations: SMF, MIL, and SIR. To apply the selected transformations on Android apps, we use the publicly available Android apps dataset by Chowdhury et al. [12] The dataset contains 24 diverse apps with 106 auto-generated test cases. We failed to build two apps, hence we conducted our experiment on 22 Android apps: 2048-Game, 24Game, AcrylicPaint, Agram, AndQuote, Bomber, Budget, Calculator, ChromeShell, DalvikExplorer, EyeInSky, Exodus, FaceSlim, GnuCash, Memopad, PaintElectric, Pinball, SensorReadout, Temaki, VLC, Wikimedia, and Yelp.

To observe each transformation's individual effect on an app, from the set SMF, MIL, and SIR, we apply each transformation separately on the apps. Furthermore, to observe the collective effect of the transformations, we apply all three transformations on each app, and refer to it as an *aggregate transformation* (AT). Altogether, we generate $22 \cdot 4 = 88$ solutions (i.e., transformed apps). To measure the energy consumption of the solutions, we ran each solution's test suite 10 times on GreenMiner [20]. Later, we compare the

ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9224-2/22/05…$15.00
https://doi.org/10.1145/3510455.3512795

energy consumption difference between the best solution (i.e., with the least energy consumption) and the original app. As a result, we observed that byte-code transformations reduced the energy consumption of 12 out of 22 apps, with an average reduction of 1.24% ± 1.96% joules, with an outlier maximum reduction of 7.1% joules for the 24Game app.

Although the results suggest that byte-code transformations affect energy, is every transformation's effect on energy consumption universal? To answer this question, for each transformation, we compare its transformed apps' energy consumption and find that each transformation affects each app differently. For example, AT reduced the energy consumption of VLC but it increased the energy consumption of 24Game. Similarly, SMF reduced the energy consumption of 24Game but it increased the energy consumption of Pinball. To find out if applying byte-code transformations have a similar effect on all the apps, on the energy consumption of the transformed apps, we applied a Wilcoxon Rank Sum Test [19] with $\alpha = 0.05$. The result implies that each transformation affects each app differently and none of the transformation holds a universal effect on energy consumption.

Since every transformation, when applied individually or collectively on an app, has a different effect, we are left with the question of: "how do we choose transformations that would reduce the energy consumption of an app?" To answer this question, we employ genetic search to find the combinations of transformations that could reduce energy consumption of an app.

3 EXPERIMENTAL SETUP

We investigate a set of 17 byte-code transformations to find energy reducing subsets. But if we apply all possible subsets of 17 byte-code transformations on an app, it would yield 2^{17} possible transformed apps. It is impractical to reliably measure the energy consumption of that many apps. As a solution, we employ genetic search to find an effective subset of transformations that could reduce energy consumption.

Genetic search is based on the principles of the theory of evolution [17, 39]. Genetic search samples from a large population of possible solutions, evolves the solutions for a new generation, and keeps sampling and evolving generations in rounds until the best solution is found [26]. We explain genetic search and how we adopt it in five steps.

1. Initial population. a search-space of 50 random solutions is generated. In our experiment, a solution is a transformed app on which certain transformations have been applied. To generate the initial population, we select apps and the transformations to apply on them.

Selecting real-world apps. Measuring energy multiple times for an app and its possible solutions is time consuming. Therefore, we choose one app from each category in our dataset. The four chosen apps include: 2048Game, a game; ChromeShell, an internet browser; AcrylicPaint, an entertainment app; and Agram, an anagram algorithm utility. The test suite available for these apps is auto-generated and it invokes multiple unique system calls.

Selecting transformations. From the set of transformations that Redex offers, we chose transformations that run separately and excluded the ones that enable or depend on other transformations. As a result, we use a set of 17 byte-code transformations (Table 1).

Table 1: A brief description of the byte-code transformations used in this study. (source: https://fbredex.com/docs/)

Type	Description
(CTP)	*Constant Propagation* Substitute the values of known constants in expressions.
(CPP)	*Copy Propagation* Replaces the occurrences of targets of direct assignments with their values.
(DCE)	*Dead Code Elimination* Removes dead code from methods.
(DSI)	*Delete Super Interface* Ensures that the method arguments pass directly through to the super invocation.
(MIL)	*Method Inlining* Inlines methods that are sufficiently small (or called only a few times).
(MRA)	*Minimum Register Allocation* Uses graph coloring to use minimum number of registers.
(PHO)	*Peephole Optimization* Eliminates redundant code patterns.
(RDB)	*Remove Duplicate Blocks* Removes duplicate blocks.
(REC)	*Remove Empty Classes* Removes classes with no-functionality.
(ROI)	*Reorder Interfaces* Reorders Interface list for each class to improve the linear walk of list.
(RBI)	*Rebind Invocations* Rebinds all invocations of a virtual method or interface to their most abstract type.
(RGT)	*Remove Gotos* Removes gotos that are chained together by rearranging the instruction blocks to be in order.
(RSM)	*Remove Synthetic Methods* Removes synthetic methods by javac.
(RUC)	*Remove Unreachable Code* Removes un-reachable methods, fields and classes.
(SIR)	*Single Interface Removal* Removes interfaces that are implemented only once.
(SMF)	*String Minification* Minify constant string literals to reduce APK size.
(SBR)	*Synthetic bridge removal* Removes bridge methods that javac creates to provide argument and return-type covariance.

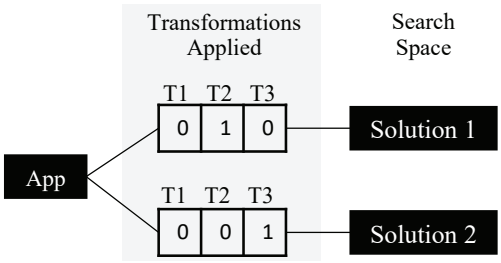

Figure 1: An example of applying two different combinations of three transformations on an app to generate a search space of two possible solutions (transformed apps).

Applying 17 transformations with their possible combinations on an app would generate 2^{17} transformed apps. To conservatively generate an initial population search-space, we start by applying 50 random combinations of 17 transformations to an app. These 50 combinations generate 50 solutions that represent the initial

population of each app. We represent each solution as a 17-bit vector that shows which transformations were applied to the solution. Each index of the bit-vector represents a specific transformation, and its value 0/1 represents if that transformation was applied or not. Figure 1 shows an example of how 3 types of byte-code transformations, when applied with two different combinations to an app, yield two different solutions (i.e., transformed apps) in a search space.

2. Fitness function. The solutions in the initial population and its successive generations are evaluated using a fitness function. A fitness function evaluates each solution's performance and assigns it a fitness score. In our case, fitness function measures the energy consumption of each solution and assigns inverse of the energy measured as a fitness score to that solution.

Energy Measurement. To accurately measure energy, we use *GreenMiner* [20], a hardware-based energy measurement tool that runs on Galaxy Nexus running Android OS 4.2.2; an Arduino Uno; and an Adafruit INA219 current sensor for monitoring power usage. We calculate the energy of the original app and its solutions in Joules (J). For accurate energy measurement, we run each solution's test 3 times. To aggregate the results, we calculate median energy consumption. Due to the error within energy measurements, median is the metric often used for aggregating energy results [20]. The fitness score of the solution is set to the inverse of the calculated energy consumption, this is to ensure a higher fitness score for lower energy consumption.

3. Selection. For each successive generation, a sample from the existing population is selected to develop a new generation. Depending on the selection technique adopted, solutions can be selected on the basis of their fitness scores, or solutions can be randomly sampled from the population.

For selection, we applied the elitist selection strategy [2] and the tournament selection strategy [7]. Elitist selection allows to pass a number of solutions with the best fitness score to the next generation. For elitist, we kept its value to 2 solutions, i.e., forwarding 2 best fitness score solutions in population to the next generation. In tournament selection, multiple solutions are sampled randomly from the population, and the solution with the highest fitness score is selected. We apply tournament selection twice and select two solutions, a pair, to perform the next step in genetic search.

4. Crossover and mutation. This process increases diversity in the search-space.

Crossover. We apply uniform crossover [15] among the previously selected pair of solutions. Using each half of the bit-vector values from each solution in the pair, two new solutions are generated and added to the next generation.

Mutation. Mutation is applied, with a certain probability, on each solution's bit-vector values. We perform mutation [21] on the population by flipping the bits of all solutions' 17-bit vectors with a probability of 1/17.

5. Termination. This step of genetic search takes place when there has not been any significant progress in the fitness score of the solutions for several generations, or if several number of generations have been evaluated without conclusion. We iterate the process of selection, mutation, and crossover until the termination criteria is reached. We terminate genetic search when either the energy has not improved in the last 10 generations or a total of 20 generations have been evaluated. We chose this termination criteria according to the cost that we can bear in terms of test execution time and energy measurement time.

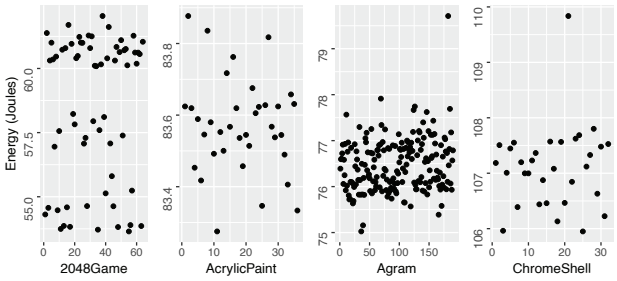

Figure 2: Optimal solutions' energy consumption in the search-space. The x-axis indexes the solutions of each app.

4 RESULTS

After running genetic search separately on four real-world apps, we get the following results.

Benefits of employing genetic search. Genetic search reduces the energy consumption for AcrylicPaint (0.42%), ChromeShell (1.15%), Agram (1.79%), and 2048Game (11.05%). Energy consumption may be reduced even further if the size of population (i.e., exploration) and the number of generations (i.e., exploitation) are increased [27]. We have also investigated if a developer may simply apply all transformations to reduce energy consumption. In Table 2, *All Transformations Solution Energy* (AT) shows that if developers apply all transformations on an app, it might in fact increase the energy consumption of an app.

Exploration effectiveness of the proposed technique. We measured the variation of energy consumption of those optimal solutions that performed better than the original app. Figure 2 shows that the standard deviation (SD) of the optimal solutions' energy consumption (joules %) is 6.91% among 234 solutions of 2048Game, 0.31% among 116 solutions of AcrylicPaint, 1.32% among 637 solutions of Agram, and 0.95% among 106 solutions of ChromeShell. The SD values show how much the energy consumption of the optimal solutions varied in the search-space. For further inspection, the exploration effectiveness of our technique may be determined by comparing the SD of the optimal solutions' energy consumption with another evaluation of the proposed technique that has different population size, genetic operators, or termination criteria.

Cost effectiveness of the proposed technique. Table 2 presents the estimated cost in terms of time consumption to find the most energy optimal solution for an app. In the table, *Test suite exec.* represents the total time to run a solution's test suite. For reliable energy measurement, we executed each solution's test suite 3 times. *Solutions explored* are the number of solutions explored in the search-space. *Analysis time* represents the overall time (actual cost of our technique) in hours to explore all the generations of the search space,

Table 2: A comparison between the energy consumption of Apps' original version and their best solution. Energy improvement shows the energy optimized using genetic search. Test suite execution is the time to run a test suite. Solutions explored are the number of solutions explored in genetic search. Analysis time shows the overall time spent for genetic search.

Apps	Original App Energy (Joules)	Best Solution Energy (Joules)	All Transformation Sol. Energy (Joules)	Energy Reduction (%)	Test suite exec. (s)	Solutions explored	Analysis time (hrs)
2048Game	60.31	53.64	55.52	11.05%	60	234	11.7
AcrylicPaint	83.62	83.27	83.74	0.42%	95	116	9.2
Agram	76.39	75.02	crashed	1.79%	77	637	40.9
ChromeShell	107.18	105.95	crashed	1.15%	100	106	8.9

Table 3: Difference in size of the original app and its best solution. *Overall Diff.* includes resource, binary code, and meta files, while classes.dex includes binary code.

Application	Overall Diff.	Diff. in classes.dex
2048Game	+0.37%	-23.46%
AcrylicPaint	-0.07%	-0.66%
Agram	-0.16%	+0.30%
ChromeShell	0.02%	-1.26%

which includes test suite execution, energy measurement and result aggregation. The overall cost of running genetic search on the selected apps is 9–41 hours. This cost is quite reasonable given the fact that reducing energy consumption would be the final step in the release process.

5 RELATED WORK

Genetic algorithms have been employed to reduce energy consumption of smart buildings [42], wireless networks [23], subway trains [8], grid systems [25], and also smartphone apps.

For software, genetic search has previously been used to find energy efficient libraries [10, 30]. Genetic search has been used to address matrix multiplication energy tuning [4], energy efficient SAT solver [9], and energy efficient assembly code exploration [38]. The closest work is by Georgiou et al. [16], Pallister et al. [33], in which they found out the effect of the GCC and LLVM compiler configurations on energy consumption. However, since their approach was evaluated on embedded systems' benchmarks, their results can not be generalized over real-world Android apps running on a smartphone device.

6 DISCUSSION

Out of 17! possible combinations of byte-code transformations some combinations reduced energy consumption. To investigate the reasons of this reduction effect, we study the effect of byte-code transformations on the structure of the apps.

To analyze the difference between the best solution and the original app, we used APK Analyzer [1]. Table 3 shows the size difference between the best solution and the original app. The overall difference includes the binary code, resource files, and meta files for the app. The difference in classes.dex indicates the size difference in the binary code. Classes.dex also keeps the code's structural information.

To further investigate the structural difference, we extract Chidamber and Kemerer object-oriented metrics (CKJM metrics) [40] from classes.dex. In literature, a sub-set of CKJM metrics have proven to have a correlation with energy consumption [37]. We used CKJM tool [40] to measure the difference between the original app and its best solution for those CKJM metrics that have strong correlation with energy, such as Depth of Inheritance Tree (DIT), Number of Immediate Sub-classes (NOC), Coupling between Object (CBO) and Afferent Coupling (Ca). For 2048Game, we observed the following changes: DIT +28.3%, NOC -100% and CBO +0.3%. For ChromeShell, we observed the following changes: CBO +1% and Ca +1.62%. For AcrylicPaint and Agram, only those metrics got affected that do not have any correlation with energy. DIT has a positive correlation while NOC, CBO and Ca have negative correlation with energy consumption [37]. Therefore, for ChromeShell, the change in CBO and Ca might be one of the reasons for energy reduction. However, for the remaining apps, the reasons of energy reduction remain inconclusive and require further investigation.

7 FUTURE PLANS

In the future, a multi-objective study can be carried out to reduce energy consumption and app size while improving run-time performance simultaneously. To improve our technique, an automated approach can be adopted to keep a balance between exploration and exploitation in genetic search [27]. The exploration speed can be improved by adopting a software based energy estimation model [11]. Other metrics like CKJM can be compared to find out what else made byte-code transformations reduce energy consumption. Furthermore, to save developers time, our collected dataset [5] that includes transformed apps' binary code, structural information, and energy measurements, can be used to train a model that could predict the best transformations for an app.

8 CONCLUSION

Byte-code transformations affect the energy consumption of Android apps, but they do not have a universally positive effect. We show that applying all byte-code transformations on an app could increase its energy consumption. To find the combination of transformations that reduces energy of an app, we presented a technique for selecting byte-code transformations using genetic search. The proposed approach have reduced the energy consumption of 4 real-world apps by up to 11%.

Using the proposed approach, developers do not have to modify the code structure or user interface of their apps. Developers can instead focus on the functional requirements, without worrying about the energy related technical details of their apps.

REFERENCES

[1] Android. 2021. Android APK-Analyzer. https://developer.android.com/studio/debug/apk-analyzer

[2] Shumeet Baluja and Rich Caruana. 1995. Removing the genetics from the standard genetic algorithm. In *Machine Learning Proceedings 1995*. Elsevier, 38–46.

[3] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated re-factoring of android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 139–150.

[4] Tania Banerjee and Sanjay Ranka. 2015. A genetic algorithm based autotuning approach for performance and energy optimization. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 1–8.

[5] Bangash. 2021. Dataset for Byte-code level Search Based Energy Optimization. https://github.com/AbdulAli/EnergyDataset-ICSE-NIER22

[6] Abdul Ali Bangash, Daniil Tiganov, Karim Ali, and Abram Hindle. 2021. Energy Efficient Guidelines for iOS Core Location Framework. In *Proceedings of the 2021 International Conference on Software Maintenance and Evolution (ICSME)* (2021-06-15). 1–12. http://softwareprocess.ca/pubs/bangash2021ICSME-igreenminer.pdf

[7] Tobias Blickle. 2000. Tournament selection. *Evolutionary computation* 1 (2000), 181–186.

[8] Morris Brenna, Federica Foiadelli, and Michela Longo. 2016. Application of genetic algorithms for driverless subway train energy optimization. *International Journal of Vehicular Technology* 2016 (2016).

[9] Bobby R Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1327–1334.

[10] Nathan Burles, Edward Bowles, Alexander EI Brownlee, Zoltan A Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-oriented genetic improvement for improved energy consumption in Google Guava. In *International Symposium on Search Based Software Engineering*. Springer, 255–261.

[11] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering* 24, 4 (2019), 1649–1692.

[12] Shaiful Alam Chowdhury and Abram Hindle. 2016. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 49–60.

[13] Shaiful Alam Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei. 2019. GreenBundle: an empirical study on the energy impact of bundled processing. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1107–1118.

[14] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

[15] Emanuel Falkenauer. 1999. The worth of the uniform [uniform crossover]. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Vol. 1. IEEE, 776–782.

[16] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. 2018. Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/3207719.3207727

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

[18] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 225–236.

[19] Winston Haynes. 2013. *Wilcoxon Rank Sum Test*. Springer New York, New York, NY, 2354–2355. https://doi.org/10.1007/978-1-4419-9863-7_1185

[20] 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 *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 12–21.

[21] Tzung-Pei Hong and Hong-Shung Wang. 1996. A dynamic mutation genetic algorithm. In *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No. 96CH35929)*, Vol. 3. IEEE, 2000–2005.

[22] Facebook Inc. 2021. Redex - An Android Bytecode Optimizer. https://fbredex.com/

[23] Sunil Kr Jha and Egbe Michael Eyong. 2018. An energy optimization in wireless sensor networks by using genetic algorithm. *Telecommunication Systems* 67, 1 (2018), 113–121.

[24] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. What Do Mobile App Users Complain About? *IEEE Software* 32, 3 (2015), 70–77. https://doi.org/10.1109/MS.2014.50

[25] Joanna Kołodziej, Samee Ullah Khan, Lizhe Wang, Aleksander Byrski, Nasro Min-Allah, and Sajjad Ahmad Madani. 2013. Hierarchical genetic-based grid scheduling with energy optimization. *Cluster Computing* 16, 3 (2013), 591–609.

[26] John R Koza, David Andre, Martin A Keane, and Forrest H Bennett III. 1999. *Genetic programming III: Darwinian invention and problem solving*. Vol. 3. Morgan Kaufmann.

[27] Lin Lin and Mitsuo Gen. 2009. Auto-tuning strategy for evolutionary algorithms: balancing between exploration and exploitation. *Soft Computing* 13, 2 (2009), 157–168.

[28] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2018. Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 14 (Sept. 2018), 47 pages. https://doi.org/10.1145/3241742

[29] 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 *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 143–154.

[30] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 503–514.

[31] Wellington Oliveira, Renato Oliveira, and Fernando Castor. 2017. A study on the energy consumption of android app development approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 42–52.

[32] Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. 2021. Improving energy-efficiency by recommending Java collections. *Empirical Software Engineering* 26, 3 (2021), 1–45.

[33] James Pallister, Simon J Hollis, and Jeremy Bennett. 2015. Identifying compiler options to minimize energy consumption for embedded platforms. *Comput. J.* 58, 1 (2015), 95–109.

[34] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. 2015. What do programmers know about software energy consumption? *IEEE Software* 33, 3 (2015), 83–89.

[35] Gustavo Pinto and Fernando Castor. 2017. Energy efficiency: a new concern for application software developers. *Commun. ACM* 60, 12 (2017), 68–75.

[36] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 22–31.

[37] Hareem Sahar, Abdul A Bangash, and Mirza O Beg. 2019. Towards energy aware object-oriented development of android applications. *Sustainable Computing: Informatics and Systems* 21 (2019), 28–46.

[38] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler software optimization for reducing energy. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 639–652.

[39] J Maynard Smith. 1978. Optimization theory in evolution. *Annual review of ecology and systematics* 9, 1 (1978), 31–56.

[40] D. Spinellis. 2005. Tool writing: a forgotten art? (software tools). *IEEE Software* 22, 4 (July 2005), 9–11. https://doi.org/10.1109/MS.2005.111

[41] Mian Wan, Yuchen Jin, Ding Li, and William G. J. Halfond. 2015. Detecting Display Energy Hotspots in Android Apps. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102585

[42] Thomas Wortmann, Christoph Waibel, Giacomo Nannicini, Ralph Evins, Thomas Schroepfer, and Jan Carmeliet. 2017. Are genetic algorithms really the best choice for building energy optimization?. In *Proceedings of the Symposium on Simulation for Architecture and Urban Design*. 1–8.