Assisting Non-Specialist Developers to Build Energy-Efficient Software

Benito Fernandes
Federal University of Pernambuco
Recife, PE, Brazil
jbfan@cin.ufpe.br

Gustavo Pinto
Federal Institute of Pará
Belém, PA, Brazil
gustavo.pinto@ifpa.edu.br

Fernando Castor
Federal University of Pernambuco
Recife, PE, Brazil
castor@cin.ufpe.br

***Abstract*—In this paper we introduce CECOTOOL, a tool that analyzes the energy behavior of alternative collection implementations and provides potentially useful recommendations about good implementation options. We applied it to two real-world software systems from the DaCapo suite [1], Xalan and Tomcat. With no prior knowledge of the application domains, we were able to reduce the energy consumption up to 4.37%.**

I. INTRODUCTION

Nowadays, developers are interested in building more energy-efficient software, even in scenarios where energy is not an obvious concern, such as desktop applications [2]. One consequence of this widespread interest in energy is that many developers who are not specialists also want their applications to consume less energy. These developers have ample knowledge and experience with their development platforms of choice, but lack both knowledge and tools when it comes to making applications energy-efficient [3].

Every non-trivial software system has parts where it is possible to use different data structures, API calls, and concurrency control mechanisms by means of simple source code transformations. We call these parts *energy variation hotspots* when these transformations reduce energy consumption. For example, the Java language has 9 different implementations of hash tables. Previous work [4] has shown that an insertion operation in one implementation, `ConcurrentHashMap`, can consume less than 1/3 of the same operation in another implementation, `Hashtable`. This also applies to other languages [5], [6], types of constructs [5], and usage scenarios [7].

In this paper, we share our vision about tools for analyzing the energy behavior of alternative implementations in an application-independent way. Based on how applications use these implementations, they can make recommendations about the most efficient implementation option, allowing non-specialists to reduce the energy footprint. We have instantiated this approach in a tool named CECOTOOL[1]. We applied CECOTOOL to two real-world software systems, XALAN and TOMCAT. With no prior knowledge of the application domains, we were able to reduce the energy consumption up to 4.37%.

Related Work. Some studies propose techniques [8], models [9], and tools [10] to mitigate the impact of energy consumption on software evolution. More related to this study, there are works that focus on programming abstractions familiar to developers, such as data structures [4], [7] and concurrency control mechanism [5], [11]. These studies share a common finding: simple changes can reduce energy consumption considerably. However, most of these studies do not provide tool support for developers. In contrast, our approach focus on non-expert developers that do not have the knowledge neither the time or tools to understand the energy impact of energy-hotspots, but still want to leverage energy savings.

[1]Stands for Collections Energy Consumption Optimization tool.

II. OUR APPROACH

Figure 1 provides an overview of our approach. The three phases are detailed next.

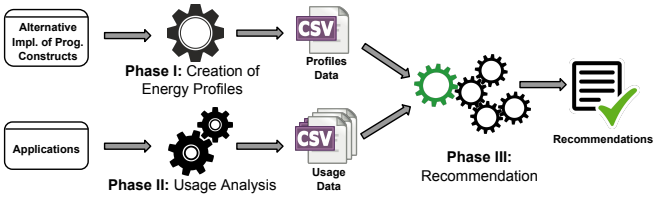

Fig. 1: An overview of our approach. Phase I is application-independent whereas Phases II and III also use information about the system under analysis.

Phase I: Creation of Energy Profiles. Here we select a group of programming constructs to analyze. We also construct their energy profiles. Good choices are constructs that are used intensively and that have alternative implementations. Having selected the candidate constructs, it is necessary to collect the energy profiles [7] of their alternative implementations. In this step, a number of benchmarks are executed to collect information about the energy behavior of these implementations in an application-independent way. This step needs only to be performed once for a given construct, per execution platform. The results can be reused across multiple software systems employing these constructs.

Phase II: Usage Analysis. This phase extracts information about how the target system uses the selected programming constructs, for example, usage context and frequency of use. This information can be extracted dynamically or statically. In our instantiation of this approach, we have used a purely

static approach. This has the advantage of being platform-independent and not requiring multiple executions of the system under analysis.

Phase III: Recommendation. This phase combines the energy profiles and the results of the usage analysis. Different formulae can be employed in this phase. A straightforward approach is to linearly combine the energy profiles with the frequency of use of the operations of each alternative implementation. Each of these combinations will yield an energy consumption number that can be directly compared to determine the most energy-efficient alternative. This is the approach we employed in our experiments.

III. Instantiation

Our preliminary instantiation focuses on energy variation hotspots stemming from usage of thread-safe collections. This approach is implemented by the CECOTOOL.

Phase I. In this phase, we build the energy profiles. These profiles are based on implementations and operations of three groups of collections: lists, maps, and sets. CECOTOOL measures the energy consumed by insertions, removals, and traversals on the following 11 alternative implementations of these collections. The output of this phase is a set of energy consumption measurements that are used in Phase III.

Phase II. CECOTOOL uses an inter-procedural dataflow static analysis to gather information about the usage of the collections. It collects information about the frequency of use and the context where the operations are invoked. We use an inter-procedural static analysis to estimate the frequency of use of the three analyzed operations.

Phase III. CECOTOOL makes its recommendation based on a simple formula that accounts for the energy profile information, the number of occurrences of the data structure operations in the source code, and whether those occurrences appear within loops or not. CECOTOOL makes its recommendation based on the result of this formula. It will recommend the data structure implementation with the lowest value.

IV. Preliminary Evaluation

Benchmarks. We used two real-world software systems from the well-known DaCapo suite [1], XALAN and TOMCAT. These software systems fit in our study because they are non-trivial, multithreaded, and use data structures intensively. Each benchmark has three workloads: small, medium, and large. We applied the large workload for both benchmarks to better simulate situations of intensive use of data structures in a concurrent environment. We set the number of threads as the maximum number of processors available: 20.

Infrastructure. We used a 2×10-core Intel Xeon (Ivy Bridge). It has 256GB (DDR3 1600MHz) of main memory and runs Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25), with Java 8. For energy measurement, we used `jRAPL` [10].

Methodology. To measure the impact of our recommendations, we created two versions of the benchmarks: the original version (before recommendation) and transformed version (after recommendation). We ran each version of each benchmark 600 times using the same JVM execution. This number of times was based on the number of samples required to obtain a 95% confidence interval. The results we report later in this section comprise the average of the remaining executions, without outliers. For statistics, we use the non-parametric Mann-Whitney-Wilcoxon (MWW) test [12] to test whether the difference among the two transformed versions is statistically significant. As for effect size, we used Cliff's Delta [13], a non-parametric effect size measure for ordinal values, and the Vargha-Delaney (A_{12}) [14] measure.

Results. When applied to XALAN, CECOTOOL recommended 13 transformations from `Vector` to `Collections.synchronizedList()`, and 10 transformations from `Hashtable` to `ConcurrentHashMap`. For TOMCAT, there was just one recommendation to switch from `Vector` to `Collections.synchronizedList()`. However, the tool presented 20 recommendations to switch from `Hashtable` to `ConcurrentHashMap`. All suggested transformations were applied. These recommendations refer to a small subset of the used data structures, since they depend on the workflow employed in Phase II.

After applying the recommendations suggested by CECOTOOL, the benchmarks were executed and the energy consumption was measured. Table I shows the results, before and after the transformation. As we can see, we were able to save 2.27% for XALAN and 4.37% for TOMCAT.

The p-value for the Mann-Whitney-Wilcoxon test was $p \leq 2.2^{-16}$ for both benchmarks. As for effect sizes, both transformed versions of XALAN (Cliff's Delta = 0.75, A_{12} = 0.88: **very large** effect size) and TOMCAT (Cliff's Delta = 0,72, A_{12} = 0,43: **medium** effect size) performed significantly better than the original ones. These statistical results confirms our experimental results: the transformed versions are more energy-efficient than original ones.

TABLE I: The Overall Results

Benchmark	Original (J)	Transformed (J)	Reduction (%)
XALAN	515.49	503.80	2.27%
TOMCAT	217.81	208.30	4.37%

V. Conclusion

In this paper we present our proposal to assist non-expert developers to improve the energy-efficient of their software without the need to understand how particular energy variation hotspots behave. Although general, we instantiate our approach in a tool targeting a common energy variation hotspot, thread-safe collections in the Java language. By following the recommendations of our tool, we were able to reduce energy consumption up to 4.37% in two non-trivial software systems.

Acknowledgements. We would like to thank the anonymous reviewers for helping to improve this paper. This research was partially funded by CNPq (304755/2014-1 and 406308/2016-0), FACEPE (APQ-0839-1.03/14), FACEPE PRONEX (APQ 0388-1.03/14), and PROPPG/IFPA.

REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.

[2] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *ICSE*, 2016, pp. 237–248.

[3] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *MSR*, 2014, pp. 22–31.

[4] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java thread-safe collections," in *ICSME*, 2016.

[5] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "Haskell in green land: Analyzing the energy behavior of a purely functional language," in *SANER*, 2016, pp. 517–528.

[6] W. Oliveira, W. Torres, F. Castor, and B. H. Ximenes, "Native or web? A preliminary study on the energy consumption of android development models," in *SANER*, March 2016, pp. 589–593.

[7] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *ICSE*, 2016, pp. 225–236.

[8] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *GECCO*, 2015, pp. 1327–1334.

[9] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia, "The power of system call traces: Predicting the software energy consumption impact of changes," in *CASCON*, 2014, pp. 219–233.

[10] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *FASE*, 2015, pp. 316–331.

[11] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *OOPSLA*, 2014, pp. 345–360.

[12] D. Wilks, *Statistical Methods in the Atmospheric Sciences*, ser. Academic Press. Academic Press, 2011.

[13] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, Nov. 1993.

[14] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal on Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.