jStanley: Placing a Green Thumb on Java Collections[*]

Rui Pereira
HASLab/INESC TEC, Universidade do Minho
Portugal
ruipereira@di.uminho.pt

Pedro Simão
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa
Portugal
p.simao@campus.fct.unl.pt

Jácome Cunha
NOVA LINCS, Universidade do Minho
Portugal
jacome@di.uminho.pt

João Saraiva
HASLab/INESC TEC, Universidade do Minho
Portugal
saraiva@di.uminho.pt

ABSTRACT

Software developers are more and more eager to understand their code's energy performance. However, even with such knowledge it is difficult to know how to improve the code. Indeed, little tool support exists to understand the energy consumption profile of a software system and to eventually (automatically) improve its code.

In this paper we present a tool termed *jStanley* which automatically finds collections in Java programs that can be replaced by others with a positive impact on the energy consumption as well as on the execution time. In seconds, developers obtain information about energy-eager collection usage. *jStanley* will further suggest alternative collections to improve the code, making it use less time, energy, or a combination of both. The preliminary evaluation we ran using *jStanley* shows energy gains between 2% and 17%, and a reduction in execution time between 2% and 13%.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software design tradeoffs**; **Software evolution**;

KEYWORDS

Energy Efficiency, Green Software, Energy-aware Software, Java Collection Framework, Eclipse Plugin

ACM Reference format:
Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of ASE 2018, Montpellier, France, September 3–7, 2018,* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[*]This work is financed by the European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718. The first and second author are sponsored by grants: SFRH/BD/112733/2015 and PTDC/EEI-ESS/5341/2016, respectively.

1 INTRODUCTION

Originally, computer performance was the principal concern for software developers and manufacturers alike. Now, while yet not a dramatic change, there has been a major shift in focus. Increasing energy costs related to ICT in organizations [11], data centers [9], and society's environmental concerns, are starting to change the way both computer manufacturers and software engineers are responding and looking at the issue of software energy efficiency.

While there are many energy estimation and measurement tools [2, 5–7, 15], these do not supply developers with the counseling needed for energy-aware development. In fact, recent findings [16, 19, 23] have shown that programmers are aware of the energy consumption problem, many times seeking help in resolving this. There are many misconceptions within the programming community as to what causes high energy consumption, how to solve these issues, and expressed heavy lack of support and knowledge for energy-aware development.

Pinto and Castor [22] argue that there are two main problems in regards to energy efficient software development: *the lack of knowledge the lack of tools*. They also mention several key points on energy-related solutions to software engineering, three of these being: static analysis tools, refactoring, and data structures.

In this paper, in Section 2, we detail our *jStanley* tool, which touches on the prior three points. *jStanley* is a static analysis tool which suggests a more energy efficient (and/or performance efficient) Java collection, by statically detecting collections used in a Java project, and which methods are used for each collection. Using this information, it not only suggests a better alternative, but can automatically change the code with the new collections if the programmer chooses so. However, it is not possible to always guarantee the change to best collection is a refactoring as some collections for instance change the order of the elements.

More specifically, this tool is based on previous work on the influence of the Java Collection Framework (JCF) in regards to energy consumption [21], where energy consumption and performance profile tables were attributed to each of the JCF's collections (Lists, Sets, and Maps), down to a method level.

We have ran an initial evaluation with 7 publicly available Java projects used in other research works and, using *jStanley*, improved the energy consumption between 2% and 17%. The execution time has also decreased between 2% and 13% (Section 3).

We conclude our paper with Section 4 describing related work, and Section 5, summarizing our paper and looking at future work.

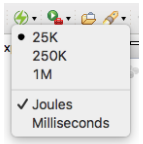
2 JSTANLEY

jStanley[1] is a static analyzer developed as an Eclipse plugin since this is the most used IDE for Java. The tool we propose is capable of statically detecting the usage of energy-inefficient collections and suggest better alternatives. It can do the same, but considering the execution time or both energy and time at the same time. To this end, it uses information on the energy consumption and execution time for Java collections [21]. This information can be provided to the tool through a set of CSV files, one per type of collection (map, list and set). Each file must contain information about the energy and time usage for each method of the collection.

jStanley constructs the abstract syntax tree (AST) of the program being analyzed, traversing it to compute the number of method calls of a given collection variable. As a result, the tool knows how many method calls for each variable of a collection the program has. For instance, for a given program, the tool can tell variable `a` of type `ArrayList` has 3 calls to the method `add` and 9 to `get`, and variable `b` of type `HashMap` has 20 calls to `put`, throughout the whole program.

Our tool provides a drop down menu within Eclipse, as shown in Figure 1, allowing programmers to select their preferences. This menu displays options to choose if they wish to focus on energy, execution time, or both, by choosing Joules, Milliseconds, or both, respectively. Additionally, as we have seen in our previous study [21], different population sizes bring about different energy profiles for collections. Thus, there is also an option to allow the programmer to choose which of the three options are closest to what they believe would best represent the program.

Figure 1: *jStanley* Eclipse menu



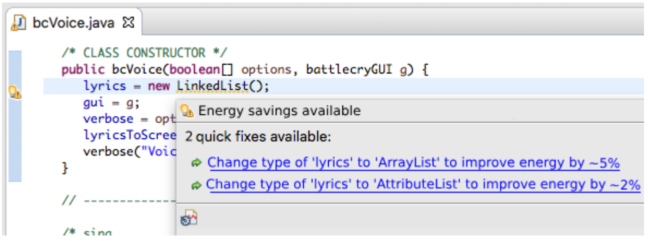
Before analyzing a program, the tool loads the data tables (from CSV files) corresponding to the settings chosen by the programmer. This information is first normalized by sorting each method from each collection and attributing a value of 1 to the lowest. Afterwards, each other value is divided by the lowest, obtaining a value greater than 1 indirectly encoding the percentage of how worse that collection's method is to the lowest one. This is done for both energy and time values, allowing the tool to combine these values to obtain an overall ranking if both options are chosen to optimize.

The calculated suggestions can be shown visually through source code flagging, as shown in Figure 2. Here a small flag appears next to an identified collection which may be changed to a more optimized one, and shows the programmer the best two alternatives. If the programmer wishes, they may select the option to change the collections to the suggested one. It is to note that pure refactoring properties may not be guaranteed when changing collections, for example natural sorting in `Tree` collections, or the non-acceptance of `null` values in `HashTable` collections.

2.1 Implementation

Four tasks divide the tool's analysis. The first task, **Source Code Analysis**, detects existing collections within the program, and all the invocations on these collections. Using the constructed AST, and the *ASTVisitor* offered by the Eclipse JDT API, we can easily visit each variable and method invocation. Using the *ASTVisitor*, we collect all the AST nodes which are *FieldDeclarations*, *VariableDeclarationStatements*, and *Assignments*. These nodes allow us to determine the data type of a variable and focus on those which are collections.

Figure 2: Suggestion flagging and quick-fix


bcVoice.java

```
/* CLASS CONSTRUCTOR */
public bcVoice(boolean[] options, battlecryGUI g) {
    lyrics = new LinkedList();
    gui = g;
    verbose = opt
    lyricsToScree
    verbose("Voic
}

// ------------

/* sing
```

Energy savings available

2 quick fixes available:

Change type of 'lyrics' to 'ArrayList' to improve energy by ~5%

Change type of 'lyrics' to 'AttributeList' to improve energy by ~2%

Afterwards, we analyze all the *MethodDeclaration* nodes. Often times, collections are passed to methods as parameters. Thus, we also collect all these method references along with the type of collections passed through, allowing us to match declarations and method invocations.

Finally, all *MethodInvocation* nodes are analyzed to determine which are JCF API methods which may be invoked during the program's execution. In these cases, we divide them in two types: direct and indirect invocations. Direct invocations would be for example a line of code with `students.add();`. An indirect invocation represents methods declared throughout the program, in which a direct invocation may occur. For example, method `m1` contains a line with `students.add(); students.add();` (two direct invocations), thus `m1` would be an indirect invocation.

The result of this analysis is a list of all the existing collections within the program, all direct invocations of each collection (and their source code location), all indirect invocations, and the collections which are passed as parameters of these indirect invocations.

In the second task, **Resolve Invocations**, using the information we now have on all the direct and indirect invocations, we calculate the amount of times each method may be used. For example, `LinkedList.add();` is within method `m1`. So we know the `add()` method on `LinkedList` will be executed once. But method `m1` is invoked twice within another method, so we may assume the `add()` method may be invoked atleast twice, and not only once. Additionally, during an indirect invocation where a collection is passed as a parameter, a match is also made with our existing list of collections. This way, we may trace every possible path a collection may go through and all possible method invocations upon the collection.

Afterwards in the **Calculate Cost** task, the cost of each collection, considering all the existing invocations and data tables, is calculated. *jStanley* calculates a matrix with all the invocation costs of each collection and method by multiplying the total number of invocations of each method by its normalized value.

Finally, in **Calculate Suggestions**, the sum of all the method costs are calculated for each collection (representing either the total cost of energy, time, or both depending on the selected options).

[1] *jStanley*, and other resources for this paper, can be found at: https://github.com/greensoftwarelab/jStanley

Table 1: Metrics of the projects evaluated. These metrics were calculated using OpenClover [1]

Project	Branches	Statements	Methods	Classes	Files	Packages	LOC
Barbecue (for bar codes)	536	2,536	369	59	59	13	8,838
Battlecry (game)	534	1,800	125	12	11	1	3,343
Jodatime (time library)	5,162	13,318	3,909	242	166	7	70,872
Lagoon (web site maintenance)	1,746	4,211	646	96	81	10	16,922
Templateit (template file generator)	408	1,067	177	22	19	3	3,317
Twfbplayer (game)	684	3,307	777	135	104	12	14,682
Xisemele (XML library)	76	522	250	57	56	3	5,770

Table 2: Evaluation data for the projects

	Test Suite		Analysis		Improvement		
Project	**#Tests**	**%Coverage**	**Analysis (ms)**	**#Changes**	**%PKG (J)**	**%CPU (J)**	**%ms**
Barbecue	152	62	2735	14	5.10	5.81	1.70
Battlecry	1*	69.4	514	4	16.79	11.49	12.76
Jodatime	4221	88.5	10490	5	7.21	7.29	7.75
Lagoon	18	4	1513	7	1.55	1.77	2.05
Templateit	3	14	1019	14	6.07	6.05	3.14
Twfbplayer	57	91	3437	51	6.04	6.30	4.36
Xisemele	167	20	588	1	4.25	4.38	3.18

* Instead of unit tests, this project has a simulated execution example

3 VALIDATION

To evaluate our tool regarding the energy savings it promotes, we selected 7 Java projects which use JCF collections and have either a test suite or simulated example of the program's execution. We obtained these projects from SourceForge [18], a repository for open-source applications, and from the SF110 corpus of classes [8], a statistically representative sample of 110 Java projects.

The selected projects and some of their metrics are listed in Table 1. The projects vary from games to time libraries. The size and complexity also varies between 3.000 and 70.000 lines of code (LOC) or between 76 and 5.000 branches in the flow graph control.

We ran *jStanley*, according to a 25k population size (since we are using only unit tests which tend to not stress collections as much), on each project and obtained the list of suggested energy optimizations, which were automatically applied. For each new project version, we re-ran the test suite, obtaining exactly the same results as the original ones. This means that, considering the available tests, the changes acted as refactorings. The amount of time spent by our tool to analyze, and the number of suggested changes are found in Table 2 under the *Analysis (ms)* and *#Changes* column.

To measure the energy consumed by each project, before and after the changes, we used Intel's Runtime Average Power Limit (RAPL) [6]. RAPL is an interface provided by (modern) Intel processors to allow the access to energy and power readings. RAPL is capable of providing very fine-grained level measurements as it has already proven [10, 25]. For our study, we measured 2 RAPL domains: PKG energy consumed by an entire socket (including the core and uncore domains); PP0 energy consumed by the CPU core.

This study was executed on a laptop with Ubuntu 14.04.5 LTS, 6GB of RAM, and Intel(R) Core(TM) i5-2430 CPU @ 2.40GHz. Both the Java compiler and interpreter were versions 1.8.0_101.

We ran each project's test suite 25 times [13], and for each execution, we extracted the energy consumed in Joules (J) for both RAPL domains, and the execution time in milliseconds (ms). The number of tests and test coverage percentage are listed in Table 2 under the *#Tests* and *%Coverage* column respectively.

The energy consumption improvements are shown in Table 2. Column *%PKG (J)* and *%CPU (J)* show the energy improvement percentage relative to the original project measured by the package and CPU respectively. Column *%ms* shows the time improvement percentage. Each value is calculated as the average of the 25 executions, excluding outliers, that is, values outside of the range $[Q1 - 1.5 \times IQ, Q3 + 1.5 \times IQ]$, where $Q1$ and $Q3$ are the first and the third quartiles, respectively, and $IQ = Q3 - Q1$ [27]. Indeed it is common to remove outliers for energy measurements [4]. For instance, we know, from experience, that the first few runs of Java programs tend to spent more energy than the remaining runs [3, 20, 21].

Using *jStanley*, we were able to achieve between 2%-17% with an average of 6.7% energy savings for PKG, and between 2%-11% with an average of 6.2% energy savings for CPU. Additionally, the performance was also improved upon with an average of 5%. *jStanley* spent on average 3 seconds to analyze our projects, with the fastest and slowest spending 0.5 and 10 seconds respectively.

These values are positive, especially when compared to the little work required by the developer. Moreover, in some cases the tests' coverage was small, maybe too small to actually stress the collections enough to make the gains more evident. Also, the fact that we used unit tests instead of actual software runs may impact the results, but most likely in a negative way. A real usage of the applications would make more use of the collections thus most likely making the gains higher. In any case, building on these positive results, we will run an in depth evaluation to fully understand the potential of *jStanley*.

4 RELATED WORK

There are several recent works showing the impact data structures have on energy consumption. Pinto et al. [24] specifically studied the energy efficiency on Java's thread-safe collections, and were able to improve up to 17% energy savings by switching out collections. Others have focused on map data structures in Android [26], and offered guidelines for choosing the most appropriate choice if one is worried about energy consumption. Lima et al. [14] analyzed the energy behavior of various Haskell sequential and concurrent data structures. They too were able to show how making changes on which data structures are used can have large impacts, saving up to 60% of energy in one of their settings. Finally, they argue that tools to support developers in quickly refactoring a program to switch between such primitives can be of great help if energy is a concern.

Two very similar studies looked at the influence of the Java Collection framework on energy consumption [12, 21]. While looking at the problem in two different perspectives, the studies are very complementary and yet again show how each collection has a different energy footprint depending on what operations and methods are invoked. The tool we present in this paper is based off of the methodology presented in the latter work.

Finally, the most similar work to ours is the *SEEDS* framework [17]. This was the first automated support for optimizing the energy usage of applications by making code-level changes. A specific instantiation of this framework was presented by the authors to improve the energy consumption of projects using Java's Collections API, producing good results. While our approach is static and based off data tables, *SEEDS* is a dynamic approach which follows a trial and error method, testing each possible alternative, until the most energy efficient one is found. While this approach may find the best alternative collection, it is a time consuming analysis. Both our studies looked at Jodatime, and while both were able to improve the energy efficiency by 8%, our static code analysis took 10 seconds while their approach took 3 hours. For a second common project, Barbecue, our approach improved it by 6%, taking 3 seconds, while SEEDS improved it by 17%, but taking again 3 hours. As expected, a static approach will be faster but not generate as good fits as a dynamic analysis.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented *jStanley*, a tool capable of quickly discovering the usage of energy-inefficient Java collections. It can also suggest and automatically evolve the code with better alternatives. The initial evaluation we performed shows promising results with savings between 2% and 17%.

As future work, we will extend the available collections, consider their memory usage, and make suggestions of collections of different kinds (e.g. list to map), performing the corresponding evolution, leading to even greater savings.

REFERENCES

[1] Atlassian. 2018. OpenClover 4.2.1. http://openclover.org. (2018).

[2] Tarsila Bessa, Pedro Quintão, Michael Frank, and Fernando Magno Quintão Pereira. 2016. JetsonLeap: A Framework to Measure Energy-Aware Code Optimizations in Embedded and Heterogeneous Systems. In *Proc. of the 20th Brazilian Symposium on Programming Languages*, Fernando Castor and Yu David Liu (Eds.). Springer Int. Publishing, 16–30.

[3] Marco Couto, Tiago Carção, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages: 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*, Fernando Magno Quintão Pereira (Ed.). Springer Int. Publishing, 77–91.

[4] Luis Cruz and Rui Abreu. 2017. Performance-based Guidelines for Energy Efficient Mobile Applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, Piscataway, NJ, USA, 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

[5] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Software-based energy profiling of android apps: Simple, efficient and reliable?. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 103–114.

[6] Martin Dimitrov, Carl Strickland, Seung-Woo Kim, Karthik Kumar, and Kshitij Doshi. 2015. Intel® Power Governor. https://software.intel.com/en-us/articles/intel-power-governor. (2015). Accessed: 2015-10-12.

[7] Miguel A Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser, and Joost Visser. 2013. Seflab: A lab for measuring software energy footprints. In *Green and Sustainable Software (GREENS), 2013 2nd Int. Workshop on*. IEEE, 30–37.

[8] Gordon Fraser and Andrea Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[9] Erol Gelenbe and Yves Caseau. 2015. The impact of information technology on energy consumption and carbon emissions. *Ubiquity* 2015, June (2015), 1.

[10] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17.

[11] Robert R Harmon and Nora Auseklis. 2009. Sustainable IT services: Assessing the impact of green computing practices. In *Management of Engineering & Technology, 2009. PICMET 2009. Portland Int. Conf. on*. IEEE, 1707–1717.

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

[13] Robert V Hogg and Elliot A Tanis. 1977. *Probability and statistical inference*. Vol. 993. Macmillan New York.

[14] Luís Gabriel Lima, Gilberto Melfe, Francisco Soares-Neto, Paulo Lieuthier, João Paulo Fernandes, and Fernando Castor. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *Proc. of the 23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering*. IEEE, 517–528.

[15] Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*. Springer, 316–331.

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

[17] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework. In *Proc. of the 36th Int. Conf. on Software Engineering*. ACM, 503–514.

[18] Slashdot Media. 2018. SourceForge. https://sourceforge.net. (2018).

[19] 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.

[20] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*. ACM, 256–267. https://doi.org/10.1145/3136014.3136031

[21] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of the 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

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

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

[24] G. Pinto, K. Liu, F. Castor, and Y. D. Liu. 2016. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. (Oct 2016), 20–31. https://doi.org/10.1109/ICSME.2016.34

[25] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (2012), 20–27.

[26] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2018. Getting the most from map data structures in Android. *Empirical Software Engineering* (2018), 1–36.

[27] John W Tukey. 1977. *Exploratory data analysis*. Addison-Wesley Pub. Co.