jStanley: Placing a Green Thumb on Java Collections

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

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

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

A video can be seen at https://greensoftwarelab.github.io/jStanley.

CCS CONCEPTS
• Software and its engineering → Software performance; Software design tradeoffs; Software evolution;

KEYWORDS
Green Software, Energy-aware Software, JCF, Eclipse Plugin

ACM Reference Format:

Original work was done by Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. jStanley is an open-source Eclipse plugin that monitors the energy consumption and performance of Java collections.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09... $15.00
https://doi.org/10.1145/3238147.3240475

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\textsuperscript{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.

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 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.

 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).

\textsuperscript{1}JStanley, and other resources for this paper, can be found at: https://github.com/greensoftwarelab/jStanley

![Figure 1: JStanley Eclipse menu](image1.png)

![Figure 2: Suggestion flagging and quick-fix](image2.png)
### Table 1: Metrics of the projects evaluated. These metrics were calculated using OpenClover [1]

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

### Table 2: Evaluation data for the projects

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

* 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 $[Q1−1.5×IQ, Q3+1.5×IQ]$, 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 spend 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, which data structures are used can have large impacts, saving up to 8% 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.

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

[26] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Szymon Szewczyk. 2009. Tools 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.