The Influence of the Java Collection Framework on Overall Energy Consumption∗

Rui Pereira, Marco Couto, João Saraiva
HASLab/INESC TEC & Minho University, Portugal
{ruipereira, marcocouto, jas}@di.uminho.pt

Jácome Cunha
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal
jacome@fct.unl.pt

João Paulo Fernandes
RELEASE, Universidade da Beira Interior, Portugal
jpf@di.ubi.pt

ABSTRACT
This paper presents a detailed study of the energy consumption of the different Java Collection Framework (JFC) implementations. For each method of an implementation in this framework, we present its energy consumption when handling different amounts of data. Knowing the greenest methods for each implementation, we present an energy optimization approach for Java programs: based on calls to JFC methods in the source code of a program, we select the greenest implementation. Finally, we present preliminary results of optimizing a set of Java programs where we obtained 6.2% energy savings.

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

Keywords
Green Computing, Energy-aware Software

1. INTRODUCTION
The increasing energy costs related to ICT in organizations [9], and society’s environmental concerns, are changing the way both computer manufacturers and software engineers develop their products. While in the previous century,

∗This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013; and by FLAD/NSF under the project Software Repositories for Green Computing, ref. 300/2015. The first author is also sponsored by FCT grant SFRH/BPD/112733/2015.

...improving execution time was the main goal when developing hardware/software, and thus programming languages and their compilers were designed to produce fast systems, nowadays energy consumption is becoming the bottleneck of such systems. As a consequence, powerful libraries offered by programming languages and their compiler optimizations have to consider this new reality.

In this paper we conduct a detailed study in terms of energy consumption of the widely used Java Collections Framework (JCF) library 1. We consider three different groups of data structures, namely Sets, Lists, and Maps, and for each of these groups, we study the energy consumption of each of its different implementations and methods. We exercise and monitor the energy consumed by each of the API methods when handling low, medium and big data sets. A first result of our study is a quantification of the energy spent by each method of each implementation, for each of the data structures we consider. This energy-awareness can not only be used to steer software developers in writing greener software, but also in optimizing legacy code. In fact, we have used/validated this quantification by semi-automatically optimizing the energy consumption of a set of similar software systems.

As a second result, we statically compute which implementations and methods are used in the source code of such projects, and then look up the energy consumption data to find which equivalent implementation has the lowest energy consumption for those specific methods. Finally, we manually transform the source code to use the “greenest” implementation. Our preliminary results show that energy consumption decreased in all the optimized software systems that we tested, with an average energy saving of 6.2%.

With our work we are answering the following research questions:

• RQ 1 - Can we define an energy consumption quantification of Java data structures and their methods?
• RQ 2 - Can we use such quantification to decrease the energy consumption of software systems?

This paper is organized as follows: Section 2 contains our analysis of the energy consumption of the different Java Collection Framework implementations. In Section 3 we describe our methodology to optimize Java programs and its application to five Java programs. Section 4 presents the validity threats for our analysis. Next, we present related and...
future work (Sections 5 and 6, respectively), and in Section 7 we present the conclusions of our work.

2. TOWARDS A RANKING OF JAVA IMPLEMENTATION’S METHODS

One of our goals is to compare the energy consumption of different Java implementations of the same abstract data structures. To do this, we designed an experiment that simulates different kinds of uses of such structures. In this section we present the design, execution, and results of that simulation.

2.1 Design

Our experiment design is inspired by the one used in [15], since our analysis also considers a simple scenario of storing, retrieving, and deleting String values in the various collections.

JCF Data structures.

The most classical way to separate Java data structures is into groups which implement the interfaces Set\(^2\), List\(^3\), or Map\(^4\), respectively. This separation indeed makes sense as each interface has its own distinct properties and purposes (for example, there is no ordering notion under Sets).

In our study, a few implementations were not evaluated as they are quite particular in their usage and could not be populated with strings. In particular, JobStateReasons (Set) only accepts JobStateReason objects, IdentityHashMap (Map) accepts strings but compares its elements with the identity function, and not with the equals method.

Given these considerations, we evaluated the following implementations:

Sets ConcurrentSkipListSet, CopyOnWriteArrayList, HashSet, LinkedHashSet, TreeSet

Lists ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

Maps ConcurrentHashMap, ConcurrentSkipListMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, Properties, SimpleBindings, TreeMap, UIDefaults, WeakHashMap

Methods.

To choose the methods to measure for each abstraction, we looked at the generic API list for the corresponding interface.

From this list, we chose the methods which performed insertion, removal, or searching on the data structures, along with a method to iterate and consult all the values in the structure. In some methods (e.g. containsAll or addAll), a second data structure is needed.

Sets add, addAll, clear, contains, containsAll, iterateAll, iterator, remove, removeAll, retainAll, toArray

Lists add, addAll, add (at an index), addAll (at an index), clear, contains, containsAll, get, indexOf, iteratable, lastIndexOf, listIterator, listIterator (at an index), remove, removeAll, remove (at a index), retainAll, set, subList, and toArray

Maps clear, containsKey, containsValue, entryset, get, iterateAll, keySet, put, putAll, remove, and values

<table>
<thead>
<tr>
<th>Table 1: Test description of Set methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>addAll</td>
</tr>
<tr>
<td>clear</td>
</tr>
<tr>
<td>contains</td>
</tr>
<tr>
<td>containsAll</td>
</tr>
<tr>
<td>iterateAll</td>
</tr>
<tr>
<td>iterator</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>removeAll</td>
</tr>
<tr>
<td>retainAll</td>
</tr>
<tr>
<td>toArray</td>
</tr>
</tbody>
</table>

2.2 Execution

To analyze the energy consumption, we first implemented our data structure analysis design as an energy benchmark framework. This is one of our contributions, and can be found at github.com/greensoftwarelab/Collections-Energy-Benchmark. This implementation is based on a publicly available micro-benchmark\(^5\) which evaluates the runtime performance of different implementations of the Collections API, and has been used in a previous study to obtain energy measurements [15].

To allow us to record precise energy consumption measurements from the CPU, we used Intel’s Runtime Average Power Limit (RAPL) [5]. RAPL is an interface which allows access to energy and power readings via a model-specific register. Its precision and reliability has been extensively studied [6, 21]. More specifically, we used jRAPL [14] which is a framework for profiling Java programs using RAPL. Using these tools permitted us to obtain energy measurements on a method level, allowing us a fine grained measurement.

We ran this study on a server with the following specifications: Linux 3.13.0-74-generic operating system, 8GB of RAM, and Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz. This system has no other software installed or running other than necessary to run this study, and the operating system daemons. Both the Java compiler and interpreter were versions 1.8.0_66.

\(^{5}\)We will refer to the population size of an object as popsize.

\(^{6}\)We will refer to the size of each such structure as secondaryCol.

\(^7\)docs.oracle.com/javase/7/docs/api/java/util/Map.html

\(^8\)dzone.com/articles/java-collection-performance
Prior to executing a test, we ran an initial “warm-up” where we instantiated, populated (with the designated population size), and performed simple actions on the data structures. Each test was executed 10 times, and the average values for both the time and energetic consumption were extracted (of the specific test, and not the initial “warm-up” as to only measure the tested methods) after removing the lowest and highest 20% as to limit outliers.

2.3 Results

This section presents the results we gathered from the experiment. We highly recommend and assume the images are being viewed in color. Figures 1, 3, and 4 represent the data for our analyzed Sets, Lists, and Maps respectively. Each row in the tables represents the measured methods, and for each analyzed implementation, we have two columns representing the consumption in Joules (J) and execution time in milliseconds (ms). Each row has a color highlight (under the J columns) varying between a Red to Yellow to Green. The most energetically inefficient implementation for that row’s method (the one with the highest consumed Joules) is highlighted Red. The implementation with the lowest consumed Joules (most energetically efficient) is highlighted Green. The rest are highlighted depending on their consumption values when compared to the most inefficient and efficient implementation, and colored accordingly in the color scale.

Figures 2 is a graphical representation of the data for our analyzed Sets. The Y-Axis represents the consumption in Joules, and the X-Axis represents the various measured methods. Each column represents a specific analyzed implementation.

The CopyOnWriteArraySet implementation was discarded during the experiment execution as the tests did not finish in a reasonable amount of time. For the full representation of the data/graphs of the other two population sizes and omitted data, please consult the online appendix. From our data, we can draw interesting observations:

- Looking at the Set results for population of 25k data (shown in Fig 1) we can see that LinkedHashSet includes most of the energetically efficient methods. Nevertheless, one can easily notice that it is also the most inefficient with the addAll and containsAll methods.

- Figure 3 presents the List results for population of 25k. Both RoleUnresolvedList and AttributeList contain the most efficient methods. Interesting to point out that both of these extend ArrayList, which contains less efficient methods, and very different consumption values in comparison with these two. We can also clearly see that LinkedList is by far the most inefficient List implementation.

- In Figure 4, we can see that Hashtable, LinkedHashMap, and Properties contain the most efficient methods, and with no red methods. Interesting to note is that while the Properties data structure is generally used to store project configuration data/settings, it produced very good results for our scenario of storing string values.

- The concurrent data structure implementations (ConcurrentSkipListSet, CopyOnWriteArrayList, ConcurrentHashMap, ConcurrentSkipListMap, and the removed CopyOnWriteArraySet) perform very poorly. As such, these should probably be avoided if a requirement is a low consuming application.

- One can see cases where a decrease in execution time translates into a decrease in the energy consumed as suggested by [27]. For instance in Figure 4, when comparing HashMap and TreeSet for the get method, we see that Hashtable has both a lower execution time and energy consumption. As observed by [25, 20], cases where an increase in execution time brings about a decrease in the energy consumed can also be seen, for example in Figure 4 when comparing HashMap and Hashtable for the keySet method. As such, we cannot draw any conclusion of the correlation between execution time and energy being consumed.

- Different conclusions can be drawn for the 250k and 1m population sizes (which can be seen in our online appendix). This also shows that the energy consumption of different data structure implementations scale differently in regards to size. What may be the most efficient implementation for one population size, may not be the best for another.

3. Optimizing Energy Consumption of Java Programs

The results presented in the previous section may allow software developers to develop more energy efficient software. In this section we present a methodology to optimize, at compilation time, existing Java programs. This methodology consists of the following steps:

1. Computing which implementation/methods are used in the programs
2. Looking up the appropriate energy tables for the used implementation/methods
3. Choosing the most efficient implementation based on total energy

In the next subsection, we describe in detail how we applied this approach and how it was used to optimize a set of equivalent Java programs.

3.1 Data Acquisition

First, we obtained several Java projects from an object-oriented course for undergraduate computer science students. For this course, students were asked to build a journalism support platform, where users (Collaborators, Journalists, Readers, and Editors) can write articles (chronicles and reports), and give likes and comments. Along with these different platform implementations, we obtained seven test cases which simulated using the system (registering, logging in, writing articles, commenting, etc.). The size of users, articles, and comments varied between approximately 2,000 and 10,000 each for each different test case and each entity. These projects had an average of 36 classes, 104 methods, and 2,000 lines of code.

Next we discuss the optimization of five of those projects, where we semi-automatically detected the use of any JCF implementation (both efficient and inefficient implementations), and which were the used methods for each implementation.
3.2 Choosing an energy efficient alternative

To try to optimize these projects based on the data structures and their used methods, we looked at our data for the 25k population. We chose this one, as it is the one which is closest to the population used in the test cases (which was between 2,000 and 10,000 for each different entity).

For each detected data structure implementation, we selected the used methods, and chose our optimized data structure based on the implementation which consumed the least amount of energy for this specific case.

Figure 5 shows the data used to make our decision for the Maps of Project 1, where Hashtable was used in place of TreeMap (as Hashtable was the most efficient implementation in this scenario with 6.8J). Table 2 details the 5 Projects, their originally used data structure implementations, new implementation, and used methods for the implementations.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Original Data Structures</th>
<th>Optimized Data Structures</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TreeMap, HashSet</td>
<td>HashTable, ArrayList</td>
<td>(containsKey, get, put, values)</td>
</tr>
<tr>
<td>2</td>
<td>HashTable</td>
<td>HashTable</td>
<td>(containsKey, get, put, values)</td>
</tr>
<tr>
<td>3</td>
<td>LinkedList, HashSet</td>
<td>ArrayList, TreeSet</td>
<td>(add, iterator)</td>
</tr>
<tr>
<td>4</td>
<td>HashSet, HashMap</td>
<td>ArrayList, HashMap</td>
<td>(add, iterator)</td>
</tr>
<tr>
<td>5</td>
<td>TreeSet, LinkedHashSet</td>
<td>ArrayList, TreeSet</td>
<td>(add, iterator)</td>
</tr>
</tbody>
</table>

3.3 Pre-energy measurement setup
Now that we have chosen our energy efficient alternative, we need to change the projects to reflect this. The source code was manually altered to use the chosen implementations. Finally, we verified that the program maintained the original consistency and state by verifying if the outputs and operations produced by these two versions did not change.

3.4 Energy measurements

To measure the original, and optimized projects, we followed the same methodology detailed in Section: 2.2 Execution. We executed the seven test cases in the same server, and using jRAPL obtained the energy consumption measurements. Each test was also executed 10 times, and the average values (after removing the 20% highest and lowest values) were calculated.

3.5 Results

Table 3 presents, for each project, the energy consumption in Joules (J), and execution time in milliseconds (ms) for both the original and optimized implementations. The last column shows the improvement gained after having performed the optimized implementations for both the consumption and execution time.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Data Structures</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Optimized</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23.744583</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24.6787895</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25.0243507</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.1994425</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.314512</td>
</tr>
</tbody>
</table>

As we can see, all five programs improve their energy efficiency. Our optimization improves their energy consumption between 4.37% and 11.05%, with an average of 6.2%.

4. THREATS

The goal of our experiments was to define the energy consumption profile of JCF implementations and validate such results. As in any experiment, there are a few threats to its validity. We start by presenting the validity threats for the first experiment, that is, the evaluation of the energy consumption of several Java data structure methods. We divide these threats in four categories as defined in [3], namely: conclusion validity, internal validity, construct validity, and external validity.

4.1 JCF Implementations Profile

We start by discussing the threats to validity of the first experiment.

Conclusion Validity.

We used the energy consumption measurements to establish a simplistic order between the different implementations. To do so, we have based ourselves on an existing benchmark (although developed to measure different things). To perform the actual measurements, we used RAPL, which is known to be a quite reliable tool [6, 21]. Thus, we believe the finding are quite reliable.

Internal Validity.

The energy consumption measurements we have for the different implementations/methods could have been influenced by other factors other than just their source code execution. To mitigate this issue, for every test we added a “warm-up” run, and we ran every test 10 times, taking the average values for these runs so we could minimize particular states of the machine and other software in it (e.g., operating system daemons). Moreover, we ran the tests in a Linux server with no other software running except for the operating system and its services in order to isolate the energy consumption values for the code we were running as much as possible.

Construct Validity.

We have designed a set of tests to evaluate the energy consumption of the methods of the different JCF implementations. As software engineers ourselves, we have done the best we can and know to make them as real and interesting as possible. However, these experiments could have been done in many other different ways. In particular, we have only used strings to perform our evaluation. We have also fixed the size of the collections in 25K, 250K, and 1M. Nevertheless, we believe that since all the tests are the same for all the implementations (of a particular interface), different tests would probably produce the same relationship between the consumption of the different implementations and their methods. Still, we make all our material publicly available for better analysis of our peers.

External Validity.

The experiment we performed can easily be extended to include other collections. The method can also be easily adapted to other programming languages. However, until such execution are done, nothing can be said about such results.

4.2 Validating the Measurements

Next we present the threats to validity, again divided in four categories, for the experiment we performed to evaluate the impact of the finding of the first study when changing the implementations in a complete program.

Conclusion Validity.
Our validation assumed that each method is on the same level of importance or weight, and does not distinguish between possible gain of optimizing for one method or another (for instance, there might be more gain in optimizing for a commonly used add method over a retainAll method). Thus, the method of choosing the best alternative implementation would need fine tuning. Nevertheless, it is consistent that changing an implementation by another influences the energy consumption of the code in the same line with the results found for the implementations/methods in the first experiment.

**Internal Validity.**

The energy consumption measures we have for the different projects (before and after changing the used implementations) could have influence from other factors. However, the most important thing is the relationship between the value before and after changing the implementations. Nevertheless, we have executed each project 10 times and calculated the average so particular states of the machine could be mitigated as much as possible in the final results.

**Construct Validity.**

We used 5 different (project) implementation of a single problem developed by students in the second semester of an undergraduate in Computer Science. This gave us different solutions for the same problem that can be directly compared as they all passed a set of functional tests defined in the corresponding course. However, different kinds of projects could have different results. Nevertheless, there is no basis to suspect that these projects are best or worst than any other kind. Thus, we expect to continue to see gains/losses when changing implementations in any other kinds of software projects according to our findings.

**External Validity.**

The used source code has no particular characteristics which could influence our findings. The main characteristic is possibly the fact that it was developed by novice programmers. Nevertheless, we could see the impact of changing data structure implementations in both the good and bad (project) implementations. Thus, we believe that these results can be further generalized for other projects. Nevertheless, we intend to further study this issue and perform a wider evaluation.

**5. RELATED WORK**

Although energy consumption analysis is an area explored for the last two decades, only more recently has it started to focus on software improvement more than hardware improvement. In fact, designing energy-aware programming languages is an active area [2], and software developers claim for tools and techniques to support them in the goal of developing energy-aware development [10]. Even in software testing, researchers want to know how to reduce energy consumption and where do they need to focus to do it [12].

Studies have shown that there are a lot of software development related factors that can significantly influence the energy consumption of a software system. Different design patterns, using Model-View-Controller, information hiding, implementation of persistence layers, code obfuscations, refactorings, and the usage of different data structures [1, 11, 13, 15, 22, 23, 24, 26] can all influence energy consumption, and all are software related implementation decisions.

Some other research works are even focused on detecting excessive/anomalous energy consumptions in software, not by comparing the overall energy consumption of different implementations of the same software system, but by using tools and techniques specialized in determining the consumption per blocks of code, such as methods [4], source code instructions [11] or even bytecode instructions [7]. Those works are based on an energy consumption model: a prediction model which can relate such blocks of code with the amount of energy that they are expected to consume. The concept of energy models have been widely used, particularly in the mobile area [7, 8, 16, 17, 18].

In a more focused and concrete way, some research works also analyzed the efficiency of data structures [15]. Manotas et al. [15] built a framework which was capable of determining the gain or loss, in a global point of view, of switching from one Java collection to another. Nevertheless, we have studied the behavior of a broader set of data structure implementations, divided between the appropriate groups, different population sizes, and a larger number of operations per structure. More recently, a study of the energy profiles of java collection classes has been performed [10] with the same goals as ours. The two studies were performed independently and seem to produce similar conclusions. Nevertheless, the energy quantification is not the only and main contribution in our paper.

**6. FUTURE WORK**

There are several directions for future work. We will continue to evolve our data and perform further tests and analyses. More specifically, we will extend our tests to evaluate other population sizes, more data structures, interface specific methods, and other types of input other than Strings.

To choose the most efficient alternative implementation, we are defining a new algorithm which uses the number of occurrences of the methods, and different weights for different methods.

We are also planning to extend this work into an automatic analysis and refactoring tool plugin. This tool would detect if an energy inefficient data structure is being used, suggest an alternative energy efficient data structure, and even automatically refactor the source-code to optimize consumption.

**7. CONCLUSION**

This paper presented a detailed study of the energy consumption of the Sets, Lists, and Maps data structures included in the Java collections framework. We presented a quantification of the energy spent by each API method of each of those data structures.

Moreover, we introduced a very simple methodology to optimize Java programs. Based on their JCF data structures and methods, and our energy quantifications, a transformation to decrease the energy consumption is suggested. We have presented our first experimental results that show a decrease of 6.2% in energy consumption.
8. REFERENCES


