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,

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

[1]docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html

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, CopyOnWriteArraySet, 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 operations 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, iterator, lastIndexOf, listIterator, listIterator (at an index), remove, removeAll, remove (at an index), retainAll, set, sublist, and toArray*

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

---

[2]docs.oracle.com/javase/7/docs/api/java/util/Set.html
[3]docs.oracle.com/javase/7/docs/api/java/util/List.html
[4]docs.oracle.com/javase/7/docs/api/java/util/Map.html

Benchmark.

To evaluate the different implementations on each of the described methods, we started by creating and populating objects with different sizes for each implementation. [5]

We considered initial objects with 25.000, 250.000, and 1.000.000 elements, providing our analysis with multiple orders of magnitude of measurement. This will allow us to better understand how the energy consumption scales in regards to population size.

When a second data structure is required, we have adopted for it a size[6] of 10% the *popsize* of the tested structure, containing half existing values from the tested structure and half new values, all shuffled.

Table 1 briefly summarizes how each method is tested for the Setcollection. The tests for the other collections are similar, and their full description can be found at the online appendix for this paper[7].

Table 1: Test description of Set methods

Method	Description of the test for the method
add	add popsize/10 elements. half existing, half new
addAll	addAll of secondaryCol 5 times
clear	clear 5 times
contains	contains popsize/10 elements. half existing, half new
containsAll	containsAll of secondaryCol 5 times
iterateAll	iterate and consult popsize values
iterator	iterator popsize times
remove	remove popsize/10 elements. half existing, half new
removeAll	removeAll of secondaryCol 5 times
retainAll	retainAll of secondaryCol 5 times
toArray	toArray 5 times

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[8] 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]greenlab.di.uminho.pt/wp-content/uploads/2016/02/appendixGreens.pdf
[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 *pop-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 *Hashtable* 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.

Figure 1: Set results for population of 25k

Methods	Concurrent SkipListSet J	ms	HashSet J	ms	Linked HashSet J	ms	TreeSet J	ms
add	1.6822	87	1.7749	87	1.4917	75	1.4817	92
addAll	1.4549	93	1.4771	89	1.9335	94	1.5101	93
clear	1.4901	78	1.0586	64	1.3288	60	1.8566	73
contains	1.4213	88	2.0685	78	1.0401	76	2.0446	79
containsAll	1.8317	96	1.4000	77	2.1748	88	1.4443	89
iterateAll	1.9225	99	1.4554	92	1.2907	83	1.3844	83
iterator	1.6096	83	1.7596	75	0.9613	76	1.7239	76
remove	1.7877	78	1.2633	75	1.2458	93	1.0700	76
removeAll	1.8072	85	2.1359	77	1.9145	100	1.3920	91
retainAll	3.2607	206	2.4092	200	2.2512	199	3.2222	193
toArray	1.4789	86	1.3833	80	1.3776	79	1.6292	80

Figure 2: Set results graph for population of 25k

Methods	ArrayList J	ms	AttributeList J	ms	CopyOn Write ArrayList J	ms	LinkedList J	ms	RoleList J	ms	Role Unresolved List J	ms	Stack J	ms	Vector J	ms
add	0.9773	71	1.1510	67	1.7839	86	1.8016	86	1.4801	76	1.1865	74	1.5659	76	1.5177	69
addAll	1.3353	76	1.0492	88	1.3586	82	1.1043	88	1.6661	76	1.8672	88	1.1015	88	1.7903	73
addAlli	1.7855	86	1.6035	68	1.1789	86	1.7272	99	1.5980	81	1.2497	85	1.2962	72	1.6268	90
addI	1.7125	93	1.3849	87	1.6558	119	1.6404	96	1.2718	85	1.3124	86	1.5287	83	1.4554	86
clear	1.1284	76	1.2409	75	1.7155	68	1.6497	74	1.6705	76	1.4304	80	1.6199	73	1.0574	71
contains	2.7568	166	2.4228	165	3.1768	167	3.1552	193	2.1751	162	2.4688	164	2.0128	166	2.1558	168
containsAll	1.5993	87	1.8053	92	2.1889	92	2.2887	118	1.3244	100	1.3930	96	1.2054	89	1.5091	87
get	2.0029	83	1.1171	78	1.4918	77	2.0168	109	2.2110	81	1.6613	71	1.8956	86	1.4978	73
indexOf	1.4447	76	2.0325	84	1.5682	70	2.6289	101	1.5674	79	1.1944	81	1.8090	81	2.0788	75
iterateAll	2.0701	79	1.0473	77	1.0103	73	2.6401	107	1.3605	85	1.7822	71	1.6036	81	1.1336	87
iterator	1.4893	84	1.1589	84	1.3922	72	1.7666	108	1.9760	73	1.3300	79	2.1895	84	1.6505	83
lastIndexOf	1.7750	99	1.7666	98	2.0383	127	2.5019	127	1.8914	92	1.4211	95	1.2260	84	1.2296	96
listIterator	1.4457	76	1.6190	84	1.3737	71	2.5003	106	1.3380	80	1.5176	85	1.6354	69	1.2746	81
listIteratori	1.7356	78	1.1552	81	1.5160	77	2.1996	105	1.7588	79	1.0334	80	1.8799	85	1.7545	78
remove	1.1308	96	1.4480	85	2.1946	162	1.6924	98	1.4560	84	1.1368	85	1.2663	86	1.4973	82
removeAll	8.0905	671	7.8108	697	7.3237	666	8.3150	752	7.6148	692	7.9911	664	7.3824	654	7.1281	665
removei	1.9135	85	1.3534	92	2.2858	118	1.7174	100	1.6308	85	1.6369	89	1.5850	81	1.5486	90
retainAll	2.7037	193	2.7845	200	2.6052	198	2.5982	205	3.0973	197	2.4172	200	2.7635	242	3.4019	245
set	0.9476	64	1.5943	70	1.9669	110	2.0474	112	1.5249	76	1.2312	73	1.4938	75	1.4957	72
sublist	1.3108	76	1.6021	80	1.4792	80	1.8457	98	1.4910	85	1.5117	71	1.7082	75	0.9414	75
toArray	1.6418	84	1.5024	84	2.0934	73	1.6739	106	1.5418	79	1.7455	83	1.5694	69	2.0213	80

Figure 3: List results for population of 25k

Methods	Concurrent HashMap J	ms	Concurrent SkipListMap J	ms	HashMap J	ms	Hashtable J	ms	Linked HashMap J	ms	Properties J	ms	Simple Bindings J	ms	TreeMap J	ms	UIDefaults J	ms	Weak HashMap J	ms
clear	2.0276	94	2.2961	88	1.8395	104	1.5761	94	1.5025	97	2.0777	98	2.1401	106	1.6706	98	1.8143	105	1.9941	95
containsKey	2.3132	105	2.1693	123	2.1343	103	1.8582	94	1.8726	103	1.6018	107	1.8055	99	1.9452	100	2.3366	89	1.9675	108
containsValue	21.5611	2305	7.8032	643	8.3615	683	8.4957	765	6.1326	462	7.3755	692	7.9912	678	9.1771	847	7.9341	714	6.7072	562
entrySet	2.2878	93	2.2363	116	1.8531	108	2.1332	107	1.8362	113	1.7800	97	2.1557	102	2.1617	115	1.7087	105	1.4666	102
get	2.3106	103	1.9972	119	1.8120	102	1.4071	100	1.8252	116	1.7851	97	1.5359	100	2.2331	115	1.5252	89	1.7185	103
iterateAll	2.1041	96	1.8353	115	2.6673	100	1.5343	91	1.6462	111	1.6362	100	2.0472	116	1.9122	111	1.6574	95	1.7139	106
keySet	1.7287	95	2.4889	124	1.6813	114	2.2226	99	1.8328	103	1.4866	92	2.0630	106	2.1680	110	1.5547	99	1.8749	105
put	1.8591	104	2.2888	102	2.4628	92	1.3123	96	2.0338	108	1.7038	107	2.1646	102	1.4355	91	2.1204	93	2.5784	105
putAll	1.4147	95	2.2852	122	1.7564	100	1.5949	105	1.8608	113	1.3097	95	2.1461	112	1.8914	116	2.3094	87	2.0750	108
remove	1.8574	92	2.2131	105	1.9256	109	1.6067	97	2.2300	106	1.9660	98	2.2178	106	1.8133	101	1.6888	92	2.4103	103
values	1.8279	85	2.4690	116	2.5755	109	2.2266	94	2.0009	107	1.9120	111	2.0692	108	1.4467	105	1.6533	100	2.4628	111

Figure 4: Map results for population of 25k

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 implementa-

tions, new implementation, and used methods for the implementations.

Table 2: Original and optimized data structures, and used methods for each project

Projects	Data Structures Original	Optimized	Methods
1	TreeMap	Hashtable	{containsKey, get, put, values}
	LinkedList	ArrayList	{add, listIterator}
2	HashMap	Hashtable	{containsKey, get, put, values}
3	LinkedList	ArrayList	{add, addAll, iterator, listIterator, remove}
4	LinkedList	AttributeList	{add (at an index), iterator}
	HashMap	Hashtable	{containsKey, get, put}
5	HashMap	Hashtable	{containsKey, get, put}
	TreeSet	LinkedHastSet	{add, iterator}

3.3 Pre-energy measurement setup

Methods	Concurrent HashMap	Concurrent SkipListMap	HashMap	Hashtable	Linked HashMap	Properties	Simple Bindings	TreeMap	UIDefaults	Weak HashMap
	J	J	J	J	J	J	J	J	J	J
containsKey	2.3132	2.1693	2.1343	1.8582	1.8726	1.6018	1.8055	1.9452	2.3366	1.9675
get	2.3106	1.9972	1.8120	1.4071	1.8252	1.7851	1.5359	2.2331	1.5252	1.7185
put	1.8591	2.2888	2.4628	1.3123	2.0338	1.7038	2.1646	1.4355	2.1204	2.5784
values	1.8279	2.4690	2.5755	2.2266	2.0009	1.9120	2.0692	1.4467	1.6533	2.4628
Total	8.3108	8.9245	8.9845	6.8042	7.7326	7.0026	7.5751	7.0604	7.6356	8.7272

Figure 5: Choosing optimized Map for Project 1

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 3: Results of pre and post optimization

	Data Structures				Improvement	
Projects	Original		Optimized			
	J	ms	J	ms	J	ms
1	23.744583	1549	22.7071302	1523	4.37%	1.68%
2	24.6787895	1823	23.525123	1741	4.67%	4.50%
3	25.0243507	1720	22.259355	1508	11.05%	12.33%
4	17.1994425	1258	16.2014997	1217	5.80%	3.26%
5	19.314512	1372	18.3067573	1245	5.22%	9.26%

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 consumptions 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 undergraduation 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 [19]. 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 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

[1] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, 11(05):477–502, 2002.

[2] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

[3] T. D. Cook, D. T. Campbell, and A. Day. *Quasi-experimentation: Design & analysis issues for field settings*, volume 351. Houghton Mifflin Boston, 1979.

[4] M. Couto, T. Carção, J. Cunha, J. Fernandes, and J. Saraiva. Detecting anomalous energy consumption in android applications. In *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 77–91. Springer International Publishing, 2014.

[5] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.

[6] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[7] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS)*, pages 1–7, May 2012.

[8] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proc. of the 35th Int. Conf. on Software Engineering (ICSE)*, May 2013.

[9] R. R. Harmon and N. Auseklis. Sustainable it services: Assessing the impact of green computing practices. In *Management of Engineering & Technology, 2009. PICMET 2009. Portland International Conference on*, pages 1707–1717. IEEE, 2009.

[10] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, US, May 2016. to appear.

[11] D. Li, S. Hao, W. G. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.

[12] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. Halfond. Integrated energy-directed test suite optimization. In *Proc. of the 2014 Int. Symp. on Software Testing and Analysis*, pages 339–350. ACM, 2014.

[13] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[14] K. Liu, G. Pinto, and Y. D. Liu. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2015.

[15] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proc. of the 36th International Conference on Software Engineering*, pages 503–514. ACM, 2014.

[16] S. Nakajima. Model-based power consumption analysis of smartphone applications. In *16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013.*, 2013.

[17] S. Nakajima. Model checking of energy consumption behavior. In *Proceedings of the First Asia - Pacific Conference on Complex Systems Design & Management, CSD&M Asia 2014, Singapore, December 10-12, 2014*, pages 3–14, 2014.

[18] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[19] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[20] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 345–360. ACM, 2014.

[21] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.

[22] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 55–61. IEEE, 2012.

[23] C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 36. ACM, 2014.

[24] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2014.

[25] A. E. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, 2013.

[26] A. Vetro', L. Ardito, G. Procaccianti, and M. Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *Proceedings of ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, 2013.

[27] T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed= compiling for energy. In *Languages and Compilers for Parallel Computing*, pages 169–184. Springer, 2014.