Energy Profiles of Java Collections Classes

Samir Hasan
Computer Science and Software Engineering
Auburn University
szh0064@auburn.edu

Zachary King
Computer Science and Software Engineering
Auburn University
zok0001@auburn.edu

Munawar Hafiz
Computer Science and Software Engineering
Auburn University
munawar@auburn.edu

Mohammed Sayagh
MCIS
Polytechnique Montreal
mohammed.sayagh@polymtl.ca

Bram Adams
MCIS
Polytechnique Montreal
bram.adams@polymtl.ca

Abram Hindle
Dept. Of Computing Science
University of Alberta
abram.hindle@ualberta.ca

ABSTRACT

We created detailed profiles of the energy consumed by common operations done on Java List, Map, and Set abstractions. The results show that the alternative data types for these abstractions differ significantly in terms of energy consumption depending on the operations. For example, an ArrayList consumes less energy than a LinkedList if items are inserted at the middle or at the end, but consumes more energy than a LinkedList if items are inserted at the start of the list. To explain the results, we explored the memory usage and the bytecode executed during an operation. Expensive computation tasks in the analyzed bytecode traces appeared to have an energy impact, but memory usage did not contribute. We evaluated our profiles by using them to selectively replace Collections types used in six applications and libraries. We found that choosing the wrong Collections type, as indicated by our profiles, can cost even 300% more energy than the most efficient choice. Our work shows that the usage context of a data structure and our measured energy profiles can be used to decide between alternative Collections implementations.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Energy Profile, Collections, API, Java

1. INTRODUCTION

Limiting energy consumption is an emerging concern in software research [8, 9, 22, 28, 30, 35, 39, 41, 42]. The scale of data centers and the limited battery lifetime of ubiquitous mobile devices have forced the owners and makers of these systems to monitor and budget for energy at all fronts—software included. Although there is a growing need for developers to optimize the energy-efficiency of their software, they are typically unaware of how to do this [34, 50].

Researchers have recently started focusing on autotuning the energy consumption inside software to optimize energy-efficiency [8, 9, 13, 14, 30]. Götz and colleagues [13, 14] contributed the initial work following the autotuning optimization approaches in performance improvement (e.g., [37]). Bunse and colleagues [8, 9] focused on adapting systems at runtime to use the most energy-efficient sorting algorithm. In recent work, Manotas and colleagues [30] designed a tool for autotuning Java applications by selecting the most energy-efficient implementations for Collections APIs.

Another approach to optimize energy-efficiency is to inform developers about the energy consequences of their high-level coding decisions, specifically to find alternative coding idioms. Researchers have explored the energy impact of design patterns [10, 29, 40] and refactoring [35, 41]. However, they were not able to provide specific guidelines, perhaps because the energy footprints of these coding decisions were too small. Manotas and colleagues [30] achieved significant energy saving only by replacing Java Collections classes, but they were not able to explain what is contributing to the improvement. Instead, they followed a search-based software engineering approach to find the alternative that produces the most energy-efficient result.

Our work focuses on creating energy profiles of popular Java Collections classes in order to guide developers. Specifically, we created energy consumption profiles of commonly used API methods for variants of three Collections datatypes: List, Map, and Set. Then, we investigated the reasons behind the difference in energy profiles by considering memory usage and bytecode executed during an operation. Finally, we explored how the Collections instances are used in real applications and how the usage impacts the overall energy consumption. Using the per-method energy profiles as building blocks, a developer can estimate the energy impact of each Collections instance and choose a more efficient alternative, if available. An essential property is that the profiles respect the constraints developers are tied to, since developers choose a Collections class on purpose, e.g., a List instead of a Set or a Map. Hence, proposing a Set or a Map to swap a List is confusing as a guideline. This is different from an autotuning approach that aggressively swaps Collections classes based only on API match [30].

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884869

This paper presents profiles for Collections types chosen from the Java Collections Framework [24], Apache Commons Collections [2], and Trove [47]. We collected the energy usage data using the GreenMiner framework [21].

Our profiles indicate how the energy consumption of these implementations varies with input sizes and data types. We noticed that for smaller lists (size < 500), the energy consumption differences are small, while the differences are more prominent if there are many elements. We found that operations that contain expensive computation tasks in bytecode traces appear to consume more energy. Surprisingly, memory usage did not seem to have an impact. In fact, a List containing integer elements is more expensive than a List containing objects; extra operations for auto-boxing contribute, rather than the memory requirement of elements.

We evaluated our energy consumption profiles by modifying the List classes in four open source Java libraries, one Java application, and one Android application. The energy measurement shows that choosing wrong Collections classes can impact energy consumption by over 300% (perhaps an extreme case), while choosing a "green" option can improve energy consumption by as much as 38%.

Our results indicate that the differences in energy consumption follow the trends suggested by our profiles. The impact is large when large collections are created during execution. This paper makes the following contributions:

- We describe a method in which the energy consumption profiles are measured on coding idioms in isolation, and are then used to provide guidance for choosing alternative coding idioms.
- We measure the energy profiles of various kinds of Collections classes obtained from different sources, and also profile energy consumption for varying input sizes and element types (Section 4).
- We explored two possible alternatives to explain energy consumption differences between operations (Section 5).
- We evaluate on real applications whether the alternative Collections classes can be swapped to predictably improve or worsen energy consumption (Section 6).

Our results are encouraging, since energy consumption profiles seem to have the potential to provide a developer guidance about choosing among Java Collections. If energy profiles of alternative coding idioms are available, developers can use them as a guideline to choose the "green" option based on their coding context.

2. COLLECTIONS CLASSES IN OUR STUDY

Java Collections classes store group of objects and provide APIs to access, modify, or iterate over the elements. Java ships with the Java Collections Framework (JCF), which provides reusable and convenient implementations of popular data structures and algorithms.

There are also many third-party implementations of similar structures. We studied two third-party implementations: Apache Commons Collections (ACC) and Trove. From ACC, we studied implementations that are alternative to those already in JCF. The Trove collections only hold primitive data types, since their goal is to reduce memory usage and improve performance (Trove requires three times less heap space than JCF implementations for larger collections [47]).

The Collections classes we studied are shown in Table 1.

Table 1: Profiled Collections Classes

Library	List	Map	Set
Java Collections Framework (JCF)	ArrayList LinkedList	HashMap TreeMap	HashSet TreeSet LinkedHashSet
Apache Collections Framework (ACC)	TreeList	HashedMap LinkedMap	ListOrderedSet MapBackedSet
Trove	TIntArrayList TIntLinkedList	TIntIntHashMap	TIntHashSet

We profiled the energy consumption of single API methods common across the implementations. More results and profiles are available at the project webpage: https://sites.google.com/site/collectionsenergy/.

3. ENERGY MEASUREMENT INFRASTRUCTURE SETUP

We used GreenMiner's hardware infrastructure to measure the actual energy consumed in joules (J) by our test programs. Each test was allowed to run to completion and the energy consumed by each test was recorded.

3.1 The GreenMiner Infrastructure

GreenMiner [21] is a hardware/software continuous testing suite. It instruments numerous devices, runs tests on these devices, and measures the energy consumption and power use of the entire device as the tests run. The GreenMiner client is a Raspberry Pi that acts as a test-bed; it controls an Android test device. The Pi executes tests on the device, and collects the results from an Arduino board that monitors the energy consumption of the test device. Energy is measured via an INA219 energy measurement chip that samples and aggregates measurements 500,000 times a second. The test-bed records and uploads the INA219 aggregate measurements to the GreenMiner webservice.

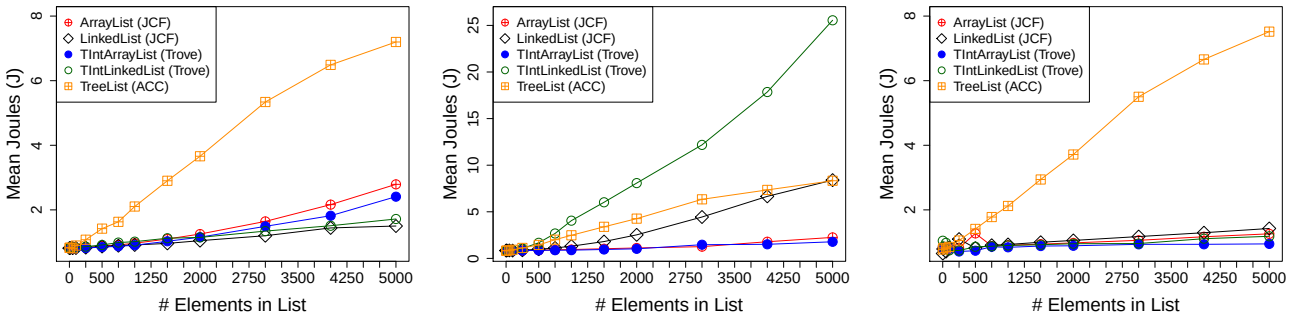
3.2 Measurement Process

To measure the effect of using different workloads on different collections, a basic Android app was created. This test-app displays a blank screen and sits idle. The screen energy consumption is constant throughout the test [11]. The test-app is a scaffold for jUnit tests to run the experiments. Each unit test for the test-app is a separate experiment or run. In each test, a Collections class was created and initialized, and a workload (insertion, iteration, etc.,) was run against it. The energy consumed by the test was measured and recorded with GreenMiner.

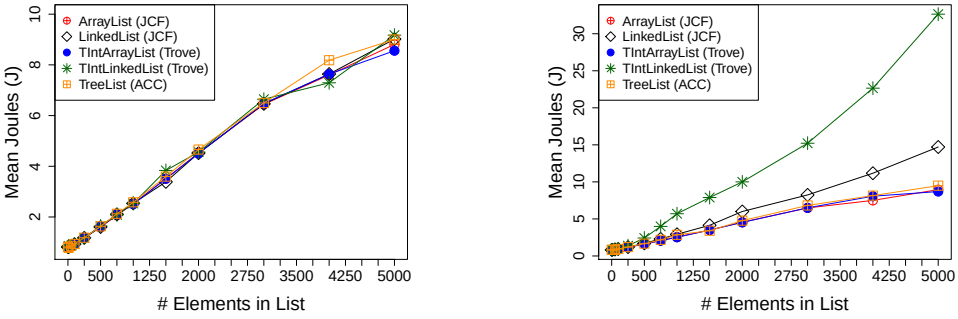
Each GreenMiner run executes unit tests for a specific use case. For example, for the use case *Insertions at the Beginning of Lists*, we wrote jUnit tests for the 5 list alternatives (Table 1). In each test, N items were added to the beginning of the list. We varied the input size N from 1 to 5000 (13 different sizes) and prepared tests for each of them. Thus, for this use case, the test device ran 65 different tests (5 kinds of lists x 13 list sizes).

Each test, given all parameters, was run 20 times on GreenMiner and the results were collected. We chose 20 measurements per test to be able to measure a 95% confidence interval and to have enough statistical power to distinguish between different energy efficiencies of the different collections.

The reports were downloaded and collated, as they report the energy consumed during each run and also the mean of



(a) Insertion at beginning (b) Insertion in middle (c) Insertion at end



(d) Iteration (e) Random access

Figure 1: Energy profiles for insertion (a) – (c), iteration (d) and random access (e) on List implementations.

20 runs. We prepared the energy consumption profiles by plotting the means against the input size N.

There were, however, a few issues with this approach. First, we needed to ensure that each unit test encounters the same overhead. Second, since our code fragments were small, their energy consumption could also be too small to be observable. Finally, the actual energy consumed by a test suite varies from device to device and the GreenMiner system is attached to 4 different devices; thus we forced all of the tests to run only on a single device. We took several measures to deal with these issues.

Ensuring a Fixed Overhead.

We created a new instance of all tested collections inside `setUp()`, irrespective of the one that is actually used for the particular test. For instance, when inserting items into a LinkedList, all the 5 list instances were first created through the `setUp()` method, followed by the actual insertions.

Producing Observable Changes.

Inside a test method, we repeated the API invocation multiple times. For example, when inserting 50 items, there were 20 runs of: (1) invoking `setUp()`; (2) inserting; (3) invoking `tearDown()`. All the unit tests were designed similarly. Thus, the numbers on our graphs are an aggregate instead of the performance of a single run. This produces an observable effect on the energy consumption of the test suite.

Ensuring Device Consistency.

We ran all our tests on a single device to remove inconsistencies. All 4 devices in the GreenMiner system use phones of the same model, but we chose to use one for all measurements to minimize differences in device-specific performance. Although each phone may report slightly different energies, the important measure here is not the absolute energy but rather the difference between two readings. As long as we use a single device, we expect the differences to be consistent.

4. ENERGY PROFILE RESULTS

We profiled the energy consumption of some of the common API methods provided by List, Map, and Set implementations, and recorded how this varies with input sizes. Specifically, we asked six research questions.

RQ1. Which List implementation is the most energy efficient for insertions, iteration, and random access?

RQ2. Which Map implementation is the most energy efficient for insertions, iteration, and random access?

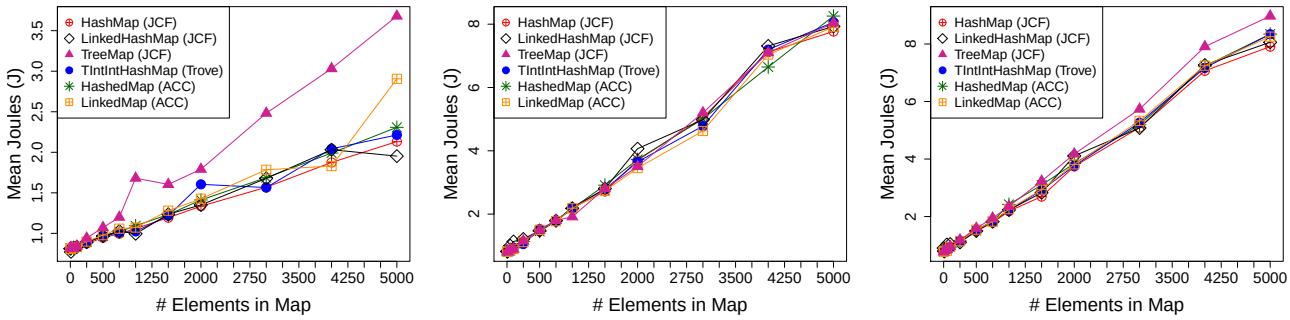
RQ3. Which Set implementation is the most energy efficient for insertions, iteration, and random access?

RQ4. How does the input size affect the energy consumption of the collections?

RQ5. How does storing different elements affect the energy consumption of the collections?

RQ6. How can we use the profiles to choose the most energy efficient implementation of List, Map and Set?

RQ1, RQ2, and RQ3 compare the energy profiles created for List, Map, and Set implementations; RQ4 and RQ5 are about measuring the impact of input sizes and data types; RQ6 is about using the results as a guideline for developers.



(a) Insertion (b) Iteration (c) Random Query

Figure 2: Energy profiles for insertion, iteration, and query on random keys in Map implementations.

RQ1. Which List implementation is the most energy efficient for insertions, iteration, and random access?

Key Result: For insertions at the beginning, JCF's LinkedList consumes the least energy, followed by Trove's LinkedList. For insertions at the middle and at the end, Trove's ArrayList is the most energy efficient, followed by JCF's ArrayList. Energy does not vary when lists expand.

Insertion

Figures 1(a), 1(b), and 1(c) demonstrate the energy consumption trends for insertion tests for List implementations.

For small sizes (1–500), the difference in energy consumption for insertions at the beginning of the list is evident only for TreeList. Even at size 250, TreeList consumes ≈ 31% more energy than ArrayList. For larger sizes, LinkedList is more efficient. At input size of 5,000, LinkedList consumes ≈ 13% less energy than TIntLinkedList, the next best performer. Compared to the worst performing TreeList, LinkedList consumes ≈ 79% less energy.

When inserting items at the middle, an interesting pattern emerges between the different list implementations. ArrayList and TIntArrayList have very similar, and quite efficient, energy performance. Next, TreeList and LinkedList both have similar, yet not quite as efficient, performance. And finally, TIntLinkedList has the worst performance by far. At input size of 500, ArrayList and TIntArrayList perform ≈ 48% better than TIntLinkedList, a large difference that increases to ≈ 93% at size 5,000. There is a substantial amount of extra energy required by TreeList, LinkedList, and TIntLinkedList to perform insertion at the middle as opposed to at the beginning.

For insertions at the end of the list, the energy differences are not obvious for input sizes below 1,000 for all lists, with the exception of TreeList, which has a noticeable degradation of ≈ 32% for size 250 (Figure 1(c)). For larger sizes, however, the differences become more evident. TIntArrayList saves ≈ 25% energy compared to ArrayList and ≈ 87% when compared to TreeList, the next best and worst energy rated lists, respectively.

We also gathered similar profiles for the case when ArrayList and TIntArrayList are not set to a predefined capacity during creation. Uninitialized array lists need to reallocate memory when current capacity is not sufficient. Due to this dynamic resizing, we expected a difference in energy consumption trends compared to the previous initialized version. However, there was no difference in the energy consumption for the uninitialized version, especially when adding items at the middle and at the end of the list. Even with dynamic expansion, ArrayList and TIntArrayList are still more energy efficient than others. Therefore, initializing an ArrayList variant with a capacity is not necessary—it will perform well anyway.

Iteration

Figure 1(d) shows the energy consumption profile for iteration with an iterator. For small sizes, iteration over an ArrayList is slightly more energy efficient—while for 5,000 items, it has a maximum energy savings of ≈ 4%. The results show that there is not much difference when comparing energy consumption of iteration over the lists.

Random Access

When accessed through randomly generated indices, we did not observe any major differences in energy consumption for list sizes smaller than 500 as shown in Figure 1(e). For larger input, ArrayList, TIntArrayList and TreeList were the most energy efficient, producing a savings of ≈ 40% compared to LinkedList and ≈ 77% when compared to TIntLinkedList.

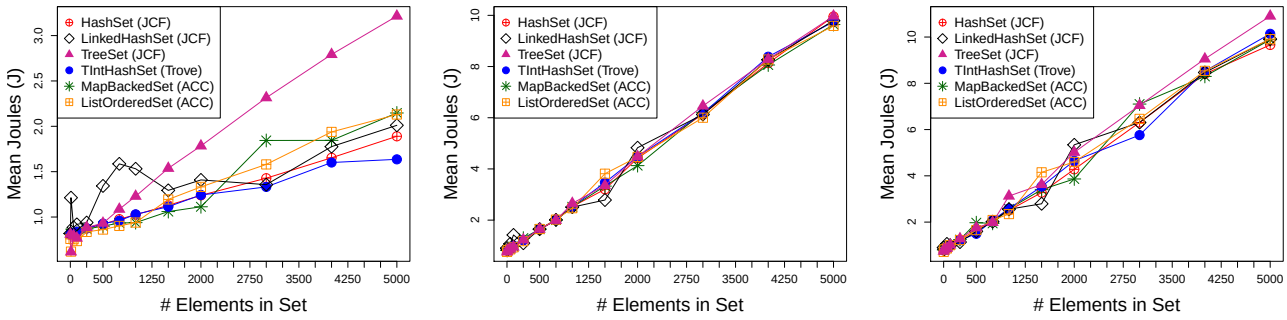
RQ2. Which Map implementation is the most energy efficient for insertions, iteration, and random access?

Key Result: HashMap is the most energy efficient alternative for insertions and random queries. If insertion order should be preserved, ACC's LinkedMap is slightly better on insertions than JCF's LinkedHashMap. TreeMap is energy hungry and should be avoided unless explicitly needed.

Insertion

Figure 2(a) shows the energy consumed by inserting key-value pairs in Map implementations. Unlike List implementations, there are some variations in energy consumption, even for smaller maps. For sizes up to 250 items, all except TreeMap perform equally well. TreeMap energy consumption increases drastically with larger input size. For 5,000 insertions, it is ≈ 73% more expensive than HashMap and ≈ 88% more expensive than LinkedHashMap.

All other maps perform equally well for sizes up to 1,000. Interestingly, HashMap performs consistently better than all other maps until size 5,000, where LinkedHashMap has a drop in energy consumption and saves ≈ 8% energy over HashMap. For most of the cases, Trove's TIntIntHashMap



(a) Insertion (b) Iteration (c) Random Query

Figure 3: Energy profiles for insertion, iteration, and query on random keys in Set implementations.

consumes more energy than HashMap. This was surprising, since the Trove implementation with primitive data types did not improve upon JCF HashMap.

Iteration

Similar to our findings for lists, the iteration performance is almost the same for all implementations (Figure 2(b)). For larger lists, JCF's HashMap requires a little less energy, while ACC's HashedMap ended up being the most expensive. However, the differences are very small for large lists, and even more so for smaller ones.

Random Query

The random query performance shows an interesting trend, as shown in Figure 2(c). For sizes up to 500, TreeMap is consistently one of the two most energy efficient maps. However, for larger lists, TreeMap queries become the most expensive, while HashMap consumes the least energy—a minimum savings of $\approx$ 2% compared to LinkedHashMap, and a savings of $\approx$ 12% when compared to TreeMap.

RQ3. Which Set implementation is the most energy efficient for insertions, iteration, and random access?

Key Result: HashSet is the most energy efficient alternative for insertions and random queries. ACC's ListOrderedSet is the most energy efficient Set for iterations, though not by a large margin. TreeSet is energy hungry and should be avoided unless explicitly needed.

Insertion

Figure 3(a) shows the energy profiles for Set insertions. For input sizes less than 750, all implementations are quite close, but for a larger size, noticeable differences arise.

Trove's TIntHashSet is consistently the most efficient, saving $\approx$ 13% energy over HashSet and $\approx$ 49% over TreeSet.

Iteration

The iteration performance is similar for all implementations (Figure 3(b))—there are no apparent differences for sizes up to 1,000. For large lists, ACC's ListOrderedSet is the most energy efficient, with a maximum energy saving of $\approx$ 4%. Again, there are often larger energy savings between smaller input sizes than there are between larger ones; for example, at size 1,500 ACC's ListOrderedSet has an $\approx$ 27% savings over JCF's LinkedHashSet, whereas the savings between the same implementations at size 5,000 are only a mere $\approx$ 2%.

Random Query

Figure 3(c) shows the energy profiles for random queries. There are many energy spikes throughout the various input sizes. Interestingly, the largest differences between implementations are in the medium size inputs, between sizes 1,000 and 3,000. Another notable trend is that TreeSet starts out in the smaller inputs to be one of the most efficient implementations for smaller sizes, then for larger sizes has an $\approx$ 13% degradation from the optimal performing HashSet. However, at size 50, TreeSet actually saves $\approx$ 4% over HashSet.

RQ4. How does the input size affect the energy consumption of the collections?

Key Result: For input sizes 1–500, all alternative implementations of List, Map, and Set perform equally well.

To determine the statistical significance of energy differences, we determined for each alternative collection and each size the 95% confidence intervals of the energy consumption measurements. For each size, we then compared the confidence intervals of each alternative collection in terms of overlap. Non-overlapping intervals indicate a significant difference between the corresponding collection types.

We found that smaller Collections with less than 500 elements do not show a significant difference in energy consumption between alternative implementations. The differences get larger and become significant as we deal with more elements. For example, for list insertions at the beginning, we compared ArrayList and LinkedList (among others) and found that for a size of 250, the confidence intervals were overlapping. However, for 500 elements, the intervals became disjoint. After comparing all other implementations in a similar way, we found the size 500 to be an appropriate threshold across all Collections types.

RQ5. How does storing different elements affect the energy consumption of the collections?

Key Result: Operations on primitive data types in lists surprisingly consume more energy than inserting small object types in a list.

When running List insertion and iteration tests on small objects, we expected the energy consumption to be higher than the lists containing primitive elements. Yet, the energy

(a) List Matrix (b) Map Matrix (c) Set Matrix

Figure 4: Color map showing the rank of each List, Map and Set implementation per use case. Green identifies an implementation as energy efficient, while red denotes it as energy hungry. Therefore, the greener the color, the better.

profiles of integer runs were consistently higher than those of the small objects. Figure 5 shows a scatterplot comparing the energy consequences of a List of integers versus a List of small objects; the results are shown for ArrayList and LinkedList. Some inputs of the ArrayList are not shown since they skew the graph. The majority of the points lie below the line y=x, whereas we were expecting all points to lie above it. For size 4,000, inserting an integer in the beginning of a JCF LinkedList is $\approx$ 13% more expensive than inserting a small object. The results are consistent with other list operations. More details and graphs are on the project webpage.

Since Java Generics do not support primitive types, the integers are auto-boxed as Integer objects in order to be held in the Collections. The lists have to add and remove this wrapper at runtime. The results are likely due to this behaviour.

RQ6. How can we use the profiles to choose the most energy efficient implementation of List, Map and Set?

Key Result: In general, TIntArrayList is the most energy efficient list implementation, followed by ArrayList. For maps, HashMap is the best, while for Sets, HashSet is the most energy efficient, with TIntHashSet as a close second.

Above the minimum input size threshold of 500 items (as mentioned in RQ4), we can use our profiles to choose the most energy efficient implementation based on the way the Collections classes are used. Figure 4 summarizes our findings as choice matrices that can help in making these decisions. Each color denotes a rank: green identifies the most efficient implementation, while red indicates the worst among the alternatives. In each table, a row with more green in it is likely to be more energy conservative on average.

In general, TIntArrayList, HashMap and HashSet (not shown) are the standout Collections implementations, followed closely by ArrayList and TIntHashSet. Lists stored as arrays are preferred in general. Linked list variants only work better if they are required to behave like a stack, i.e., a datatype with items added and removed from the front. Figure 4 may also help in finding the best Collections implementation instead of the best library, since one can easily mix-and-match different implementations of different libraries. Choosing one collection from one of the libraries does not lock developers into that library.

5. WHY THESE ENERGY DIFFERENCES?

We carried out further investigations to discover the key factors that may explain the different energy consumption

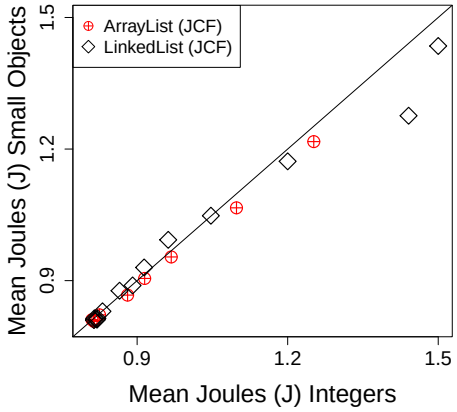


Figure 5: Comparing insertion of small objects to insertion of integers.

profiles. We explored two possible factors—(1) memory usage during API operations and (2) time-consuming bytecode instructions executed during API operations. Here, we explain the differences in energy consumed during different kinds of insertion operations (`add()` methods) on List implementations. Other Collections types and API methods are covered on our project webpage.

5.1 Memory Usage

We recorded memory consumption for List instances before and after invoking the `add()` operation, while adding 500 items to the list. We chose 500, since RQ4 showed that Collections with 500 or more items show significant differences in energy consumption.

Figure 6 shows the resulting memory usage graph when inserting at the beginning of the list. The graphs for insertions at the middle and at the end of the list were almost the same, which indicates that no matter how the items are inserted into the list, the memory footprints are similar. Yet, the energy profiles were different for the different insertion approaches (Figures 1(a)–1(c)). Therefore, memory consumption is not a (significant) driving factor behind the differences in energy consumption.

5.2 Executed Dalvik Bytecodes

We generated bytecode traces during the execution of an `add()` operation on two List instances (ArrayList and LinkedList) and compared them. First, we used dexdump to extract the application's bytecode. Then, we instrumented each line of the bytecode using the AndBug debugger tool, which implements the Java Debug Wire Protocol (JDWP). Upon execution, the tool prints out the executed bytecode and the corresponding source line.

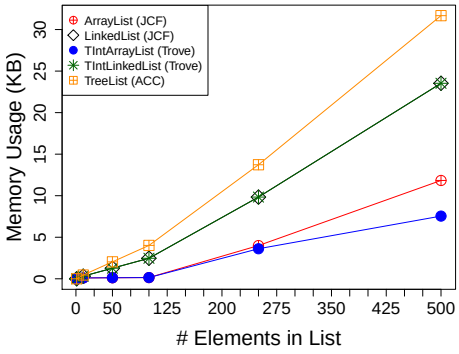


Figure 6: Memory usage of List implementations during insertion at the beginning.

Comparing the traces, we identified two bytecodes that may have an impact on the runtime (and therefore energy) performance: `iget-object` and `invoke-static`. When elements are inserted in the middle, `iget-object` is executed many more times than the other instructions. This is because LinkedList traverses half of the list to reach to the middle and locate the position for the new item. The larger the List becomes, the more traversals are needed. For example, when the 500^{th} element is inserted to a list, `iget-object` is executed 63 times more than the next frequently occurring instruction. This may explain why LinkedList consumes more energy than ArrayList, as shown in the energy profile (Figure 1(b)).

The impact of bytecodes are less obvious when elements are inserted at the beginning or at the end. When inserting at the beginning, `invoke-static` dominates the execution for an ArrayList (used to execute the expensive `System.arraycopy()` method). This difference in workload is probably why an ArrayList instance consumes more energy than a LinkedList instance for insertions at the beginning, as shown in our profiles. However, `invoke-static` is also executed while inserting elements at the end. In this case, its impact is likely offset by many other bytecode instructions only found in LinkedList execution traces (`new-instance` and `invoke-direct`). Hence, this analysis is not enough to explain why ArrayList performs better in this context. This is left as future work.

6. EVALUATION

The energy profiles compare the Collections classes for each API method and suggest better alternatives (cf. RQ6). However, when Collections instances are used in applications, multiple API methods are invoked on each object, depending on the role of the object in the system and the load of the system. Hence, we expect that the energy footprint of each Collections object in an application is determined by a combination of the energy impact of all invoked API methods. To analyze this, we ask two more research questions:

RQ7. Do the different Collections classes have an energy impact in real applications compared to what we found for similar collections in the profiles? How large is the impact?

RQ8. Can we use the energy profiles to switch to an alternative collection and improve (or degrade) the energy consumption of an application?

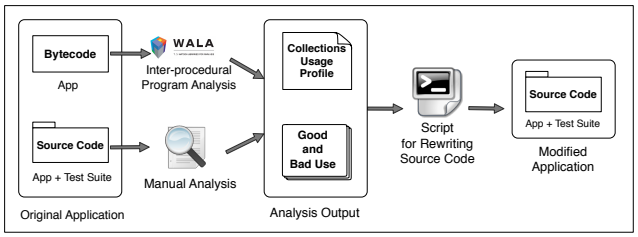


Figure 7: Evaluation workflow.

To answer RQ7, we modified real applications to use alternative Collections classes and measured the energy consumed by the modified applications. Previous work has demonstrated that Collections classes do have an impact [30]. We extend this state of the art by selectively (based on the usage profile of each instance) modifying the Collections instances using the energy profiles (RQ8): we create "good" and "bad" versions of the original program when possible, and compare their energy consumption using GreenMiner.

Using the methodology of Figure 7, we studied the energy consumption of four popular Java libraries—Google Gson [15], Apache Commons Math [4], XStream [51], Apache Commons Configuration [3]—, an open source email client, K-9 Mail [26], and a Stock Exchange Trading Simulator application. Each library came with a large test suite (16–137 KLOC). We analyzed the Collections instances used in the code to create usage profiles, i.e., to determine which API methods are being invoked and where. We wrote an inter-procedural program analyzer based on WALA [48] that automatically analyzes program bytecode.

Our WALA analyzer detected three kinds of Collections instances: (1) Collections instances declared as fields of a class and used in multiple methods, (2) Collections instances locally created inside methods and used in the same method, and (3) Collections instances locally created inside methods, but used in multiple methods since they are passed as a return value. The inter-procedural analysis uses call graphs and control flow graphs created by WALA to collect usage profiles for these instances. Currently, we do not support the analysis of Collections instances when they are passed as an argument to a method. Adding this would require another inter-procedural analysis, but we did not find enough such instances to justify the implementation. We manually analyzed these remaining instances.

For each Collections instance found by WALA or our manual analysis, we identified if it is used in an energy-appropriate manner or a better alternative is available, based on simple heuristics derived from RQ6. To create a "good" version, we looked into the usage of Collections instances to see whether swapping to another Collections class may save energy. For example, our profiles suggest that ArrayList is more energy-efficient than LinkedList when inserting items at the end of the list or when iterating over the list (these two are the most common List methods). So, if a LinkedList is used in a program for these operations, our WALA program will detect it and indicate that we can improve energy consumption by replacing LinkedList with ArrayList.

Similarly, we prepared "bad" versions by going against our profiles. For example, the profiles suggested that ArrayList is more energy efficient than LinkedList for common list operations. Instead of following this recommendation, a "bad"

version replaces ArrayList with LinkedList. We expected this change to increase the energy consumption.

Next, we used a Python script to perform lexical analysis on the source code and transform the List instances to alternatives that should improve (or degrade) the energy consumption. A simple lexical analysis was sufficient, since we swapped between alternative Collections instances with (almost) the same API (similar to Manotas et al. [30]). Furthermore, we chose to deal only with lists during our evaluation. There are two reasons behind this choice. First, changing ArrayList to LinkedList (or vice-versa) is safe—the code, if it compiles, behaves the same way irrespective of the implementation. This may not be the case if we change a HashMap to a TreeMap, since if the key object does not have an appropriate `compareTo()` method defined, the maps may behave differently. It is even more difficult to convert a TreeMap to a HashMap, since the sorting behavior of a TreeMap may be desired in a usage scenario. Second, lists are more widely used than other collections such as maps or sets (Gson: 60%, K-9 Mail: 57%, Apache Commons Math: 56%, XStream: 50%, Apache Commons Configuration: 53%, Stock Exchange Trading Simulator: 57%). Therefore, the energy contribution from lists is probably higher than that from other collections.

Eventually, we created four "bad" versions and three "good" versions. For the first three libraries, the developers almost exclusively used ArrayList whenever they needed a list data structure and followed the common usage profile of adding an item at the end of a list and/or iterating the list. Thus, we found little scope to improve on the energy consumption for these libraries. Instead, it was more interesting for those three systems to demonstrate worse energy performance by changing most of the ArrayList instances to LinkedList. In K-9 Mail, we had comparatively more linked lists, so there was a scope to make improvements. We therefore had both a "good" and a "bad" version for it. We also made "good" versions of the Apache Commons Configuration library and the Stock Exchange Trading Simulator application, since their usage of lists could be optimized for energy consumption.

The next sections describe the results of our analyses for the six applications. To address RQ7, we report the *energy impact* when we used alternative Collections instances, while for RQ8, we report the *changes that we made* (guided by our profiles) to get a "good" or a "bad" version.

6.1 Google Gson

Google Gson is a serialization/deserialization library that provides mechanisms to convert Java objects to JSON and back [15]. We used version 2.1 for our study, consisting of about 13 KLOC and a test suite of 16 KLOC. The Gson API refers to most of the collection instances through Collections interfaces, e.g., List, Map, and Set. The developer chooses whether to use an ArrayList or LinkedList instance, for example, where a List is required.

Changes Made. We found 53 ArrayList instances in the codebase (Table 2). Only 4 of these instances are part of the library code, while 49 are in the test suite. Our WALA program discovered 4 instances on which end-insertions and iterations were performed; modifying these to LinkedList instances should be a "bad" change. We studied the rest of the instances manually, and changed 47 instances to LinkedList that had the same usage pattern.

Energy Impact. Table 2 shows the percentage of change in energy consumption of the modified application. With LinkedList, the energy consumption increases by 309%. There are two factors contributing to this large increase. First, the library performs 4 times slower when LinkedList instances are used, which may cause more energy consumption during the test run. Second, the Gson test suite has a number of performance tests that perform serialization and deserialization on large inputs ($\approx$ 2-4 MB). Our profiles, as discussed in RQ4, indicate that the energy differences are more significant with larger collections, which is directly reflected through these results.

6.2 Apache Commons Math

The Apache Commons Math library provides implementations of mathematical and statistical algorithms that are otherwise unavailable in the standard Java distribution [4]. We used version 3.4.1 (209 KLOC app + 137 KLOC test). The library creates 167 instances of ArrayLists. There are 91 ArrayList instances in the test suite.

Changes Made. Since it was not possible to run all the tests on our device due to memory constraints and incompatibility issues, in our study we selected a subset of tests (71%) that compiled successfully.

Out of a total of 258 instances of ArrayList in the codebase and tests (Table 2), WALA detected 77 instances that were used mostly for end-insertions (60 occurrences), iterations (18 occurrences) and random access (5 occurrences). After manual inspection, we found 169 more instances that were used similarly. Our profiles suggest that LinkedList is a bad choice for these instances, and we used this to make a "bad" version.

Energy Impact. The modified version consumed 15% more energy (Table 2). Again, the changed library runs $\approx$ 1.2 times slower than the original version, which may have caused more energy to be consumed.

6.3 XStream

The XStream library can be used to serialize Java objects in XML and deserialize it back [51]. XStream version 1.5 has a library of 34 KLOC and a unit test suite of 30 KLOC. There are 33 ArrayList instances in the library code and 128 instances in the test code.

Changes Made. We choose a subset of the test suite (80%), as some tests were incompatible with the GreenMiner platform. Out of 161 instances of ArrayList, WALA detected 23 instances that were used for end-insertions (20 occurrences), iterations (4 occurrences) and random access (2 occurrences). We manually found another 130 instances used in the same way. In total, 153 ArrayList instances were converted to LinkedList, expecting higher energy consumption.

Energy Impact. There is a degradation of 5% when swapping the ArrayList instances with LinkedList (Table 2). The modified version runs $\approx$ 1.05 times slower than the original, which may explain why it has a higher energy consumption.

6.4 K-9 Mail

The K-9 Mail version 5.101 codebase has 34 KLOC, with a test suite of 2 KLOC. There are a total of 294 instances of collections used, out of which only 28 were covered by the test suite. To make sure more of the collections are exer-

Program	KLOC	Collections	ArrayList	LinkedList	# Changes		Changes in Energy Consumption	
					Good	Bad	Good	Bad
Google Gson	29	100	53	7	0	51	0	309% ▲
Apache Commons Math	346	461	258	2	0	246	0	15% ▲
XStream	64	324	161	1	0	153	0	5% ▲
K-9 Mail	77	294	148	21	4	125	0.25% ▼	0.32% ▲
Apache Commons Configuration	76	154	69	12	12	0	1.47% ▼	0
Stock Exchange Trading Simulator	11	14	0	8	8	0	38% ▼	0

Table 2: Libraries and applications for evaluating the energy profiles

cised, we augmented the original test suite by generating 256 more test cases for the app. We used JTExpert [43] to automatically generate tests. As this generates tests for the Java platform, and Android tests should inherit from the class *AndroidTestCase*, we modified the generated tests to adapt them to the Android platform by using JavaParser [25]. In order to know which collection method is called by the executed tests, we did a dynamic analysis using AspectJ.

Changes Made. K-9 Mail application uses 148 ArrayList instances and 21 LinkedList instances (Table 2). With our WALA analysis, we found that there was scope to prepare both a "good" and a "bad" version of the program.

Our WALA program found 53 instances of ArrayList that were used for ArrayList-friendly operations (40 occurrences of end-insertions, 9 occurrences of iterations and 2 occurrences of random accesses). We manually found 72 other instances having a similar usage pattern. We changed these instances to LinkedList, thereby creating a "bad" version.

With our WALA analysis, we also found 21 LinkedList instances in the codebase. Our heuristics suggested that we should change 4 of these instances to ArrayList, because the lists were being used for end-insertions, insertions at a random index, and queries using the `contains()` API. According to our profiles, ArrayList is the most energy efficient choice in this context. We therefore created a "good" version of the app by changing these 4 instances to ArrayList.

Energy Impact. Table 2 shows the differences in energy consumption of the two versions. For "bad" changes, K-9 Mail performed only slightly worse, with an overall degradation of 0.32%. We did expect an increase in energy consumption, although it is only by a small amount. For the "good" changes, we achieved an improvement of 0.25%.

For both versions, we noticed that the differences were very small. The K-9 Mail test suite is significantly different from the rest of the applications that we studied—it does not exercise large collections. In Gson, the tests were feeding a huge load ($\approx$ 2-4 MB) to the lists. On the contrary, the K-9 Mail tests were dealing with lists of only a few elements. Therefore, the impact was very small.

6.5 Apache Commons Configuration

The Apache Commons Configuration library facilitates storage and retrieval of configuration information for Java applications [3]. We studied version 1.10 that has 40 KLOC of library code, and a test suite of 36 KLOC. There are 13 instances of LinkedList and 166 instances of ArrayList in the original codebase.

Changes Made. We again choose a subset of the program (83%) that was compatible with the testing platform. In the reduced version, we had 69 ArrayList instances and 12 LinkedList instances. Out of these 12 LinkedList instances, WALA detected 8 that were used for end-insertion and iteration. We manually found the other 4 of them to be used in a similar way. Since our profiles indicate that ArrayList is a better choice for these operations, we changed these 12 instances to ArrayList, expecting a decrease in the energy consumption of the test suite.

Energy Impact. Changing the LinkedList instances to ArrayList improved the energy consumption by 1.47% (Table 2). The modified version of the library ran $\approx$ 1.02 times faster than the original, which is probably why the energy consumption was lower.

6.6 Stock Exchange Trading Simulator

This is a Java based simulation application developed in-house at Auburn University. The program has 11 KLOC lines of application code, with 8 instances of LinkedList used in the codebase.

Changes Made. Our WALA analysis detected 7 instances of LinkedList that were used for end-insertions and iterations. We manually found 1 more instance being used in a similar way. Since ArrayList is better for both of these operations, we made "good" changes by swapping the LinkedList instances with ArrayList.

Energy Impact. Our modified program demonstrated a 38% reduction in energy consumption and ran $\approx$ 1.6 times faster.

6.7 Discussion

For all test applications, we obtained differences in energy consumption by changing the Collections instances (RQ7). The magnitude of change, however, depends on how aggressively the instances are exercised during program execution. With significant usage, we can get large increases in energy consumption for bad choices of Collections instances (RQ8).

We also noticed that for each of the applications, the degradation factor for energy consumption was the same as the slowdown factor of the bad version of the program. For example, the version of Gson with bad changes ran $\approx$ 4 times slower and consumed $\approx$ 4 times more energy than the original version. However, the *power* consumption of both versions (i.e., the rate of energy consumption per time unit) was very similar, which indicates that the bad version (with ArrayList instances changed to LinkedList) just does more work during the *extra* time it is executing. This may explain why it consumes more energy. We found a similar trend in execution times while generating our profiles. However, to determine whether it is a consistent trend that more energy is only being consumed because of slower applications, more investigation is needed.

In our WALA analyzer, we chose a simple heuristic to help

us decide whether to change a Collections instance. This worked well owing to the fact that we dealt only with lists, and that most of the ArrayList instances that we found were performing end-insertions and iterations anyway. In the future, we want to focus on developing a more sophisticated heuristic to handle other Collections.

7. THREATS TO VALIDITY

A number of issues affect the validity of our work. First of all, measurements of physical systems, in particular phones, inherently are affected by noise and non-determinism. Our test-bed was designed to minimize such noise and to control for non-deterministic differences in measurements by repeating measurements 20 times. Section 3.2 discusses how we have addressed other measurement-related issues.

How generalizable are our results? The GreenMiner infrastructure uses Android devices to perform the energy profiling. We ran our tests on a single phone that had a specific version of the Android OS installed. We expect these energy trends to be similar across Android devices. For example, we found similar trends when we ran the tests on the other three devices on GreenMiner.

A more subtle issue may arise due to the range of the measurements that we achieved. As we saw in the profiles, the energy measurements are quite small, especially for small collections. This is expected, since a single API usage corresponds to a maximum of three lines of code performing an operation and there is a significant overhead introduced by setup and teardown methods of each test. To validate whether this large noise could overshadow the otherwise small energy consumption of a single API invocation, we ran a separate baseline test running only the setup and teardown methods, i.e., the noise. We found that each of our actual tests consumed substantially more energy than this baseline, i.e., our measurements reflect the energy contribution from the API usage.

8. RELATED WORK

There has been a large body of empirical work measuring the impact of code change in various domains. Source level modifications such as refactorings [35, 41], design patterns [1,10,29,40], and code obfuscation [42] have been found to affect an application's energy consumption. Hindle proposed Green Mining to study how changes across software versions affect energy consumption [20]. Others have worked on evaluating energy behavior of sorting algorithms [8, 9], web-servers [31], lock-free data structures [22], API usage of Android apps [28], recommending energy-efficient Android apps to users and developers [23], and the effect of advertisements and ad-blocking on energy consumption [16, 39].

Li et al. [27] repeatedly profiled Java bytecode instructions to link source code and bytecode to energy consumption in order to estimate the energy consumption of a line of Java code. JalenUnit [33] uses PowerAPI and statistical execution sampling to automatically generate benchmarks to measure the energy consumption of an API.

Researchers have measured energy in a number of ways. Hardware systems such as the Atom LEAP platform [45] and WattsUp meters [49] can measure actual power consumed by an application. Cycle accurate simulators such as SoftWatt [18], Sim-Panalyzer [32], and simplepower [52] provide an energy estimate by simulating CPU cycles for each component used in executing the application. Estimation based approaches [5–7, 12, 19, 33, 46] use empirical data to propose a model for estimating energy consumption. Pathak et al. [36] and Aggarwal et al. [5] show that dynamic analysis of running systems, specifically by extracting system calls, can produce accurate runtime models of a system and estimate the energy consumption impact of a change. Similar work on execution logs by Gupta et al. [17] fingerprinted modules for their energy consumption profile. Zhang et al. [54] describe an online profiler called PowerTutor that models energy consumption by aggregate models of individual components such as network and CPU.

Our work is closest to the approach taken by Manotas et al. [30]. The paper describes an autotuning framework, SEEDS, that aids in automatically choosing the most energy-efficient collection from the Java Collections API. SEEDS achieves this by running an exhaustive trial-and-error on all compatible collection implementations and measuring the impact of each on the overall energy consumption of a given test suite for the application. Our approach is significantly different in making the comparison. Instead of an exhaustive search on which implementation is best for the particular application (test suite), we use empirical evidence, i.e., the energy profiles that we derived in this work and the API usage patterns, to predict the best alternative.

Another closely related work is Chameleon [44], which is an autotuning approach for optimizing collection usage. However, it is particularly focused on memory usage and runtime performance (e.g., clock time), which could be a proxy for energy consumption. We believe that an autotuning framework can be equipped with our profiles to make a more accurate and realistic tool.

The motivation of all of the above work comes from studies on developer and consumer knowledge about software energy consumption that indicate that developers and consumers are not sufficiently aware of how much energy their software consumes, what are the energy bottlenecks, and which programming practices should be avoided [34, 38, 50, 53].

9. CONCLUSION

Our results provide a guideline about the scenarios in which the energy consumption of alternative Collections classes becomes an issue. For insertion operations, the energy differences are significant, but not that much for other list operations. Also, for lists of small size, the energy consumption does not vary much between the lists. Furthermore, many of the differences in energy consumption can be explained by expensive bytecode operations.

Overall, our results will be especially useful for developers of large scale software who commonly work with large Collections instances. They can guide the developers and make them aware of the consequences of their programming decisions. Our approach can also be used in making smarter autotuning tools. Most importantly, this should motivate future work on creating better guidelines for many other alternative programming choices.

More information is available on the project webpage: https://sites.google.com/site/collectionsenergy/.

Acknowledgments

We thank the reviewers for their comments. This work was supported by the NSF grant CCF-1217271 and NSERC.

10. REFERENCES

[1] S. A. Abtahizadeh, F. Khomh, and Y.-G. Guéhéneuc. How green are cloud patterns? In *Proceedings of the 34th IEEE International Performance Computing and Communications Conference (IPCCC)*, Nanjing, China, December 2015.

[2] Apache commons collections. http://commons.apache.org/proper/commons-collections/source-repository.html.

[3] Apache commons configuration. https://commons.apache.org/proper/commons-configuration/index.html.

[4] Commons math: The apache commons mathematics library. https://commons.apache.org/proper/commons-math/.

[5] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. In *Press of the 2014 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp*, 2014.

[6] N. Amsel and B. Tomlinson. Green tracker: a tool for estimating the energy consumption of software. In *CHI'10 Extended Abstracts on Human Factors in Computing Systems*, pages 3337–3342. ACM, 2010.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[8] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *MDM'09. Tenth International Conference on*, pages 600–607. IEEE, 2009.

[9] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour. Choosing the" best" sorting algorithm for optimal energy consumption. In *ICSOFT (2)*, pages 199–206, 2009.

[10] C. Bunse, Z. Schwedenschanze, and S. Stiemer. On the energy consumption of design patterns. In *EASED@ BUIS*, pages 7–8. Citeseer, 2013.

[11] M. Dong, Y.-S. K. Choi, and L. Zhong. Power Modeling of Graphical User Interfaces on OLED Displays. In *DAC 2009*, DAC '09, pages 652–657, New York, NY, USA, 2009. ACM.

[12] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th MobiSys*, pages 335–348. ACM, 2011.

[13] S. Götz, C. Wilke, S. Richly, and U. Aßmann. Approximating quality contracts for energy auto-tuning software. In *GREENS 2012*, pages 8–14, June 2012.

[14] S. Götz, C. Wilke, M. Schmidt, S. Cech, and Uwe. Towards energy auto tuning, Aug. 21 2013.

[15] Google gson. https://code.google.com/p/google-gson/.

[16] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 100–110, Piscataway, NJ, USA, 2015. IEEE Press.

[17] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran. Detecting Energy Patterns in Software Development . Technical Report MSR-TR-2011-106, Microsoft Research, 2011.

[18] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Using complete machine simulation for software power estimation: The softwatt approach. In *Proceedings of the HPCA8*, pages 141–150. IEEE, 2002.

[19] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *GREENS, 2012 First International Workshop on*, pages 1–7. IEEE, 2012.

[20] A. Hindle. Green mining: A methodology of relating software change and configuration to power consumption. *Empirical Softw. Engg.*, 20(2):374–409, Apr. 2015.

[21] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: a hardware based mining software repositories software energy consumption framework. In *Proc. of the 11th MSR*, pages 12–21. ACM, 2014.

[22] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *INTERACT*, pages 63–70. IEEE, 2011.

[23] R. S. Infantes, G. Beltrame, F. Khomh, E. Alba, and G. Antoniol. Optimizing user experience in choosing android applications. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[24] Java collections framework. http://docs.oracle.com/javase/8/docs/technotes/guides/collections/.

[25] Java parser. https://github.com/javaparser/javaparser.

[26] K-9 mail. http://k9mail.org/.

[27] D. Li, S. Hao, W. G. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 ISSTA*, pages 78–89. ACM, 2013.

[28] 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 MSR*, pages 2–11. ACM, 2014.

[29] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides. Energy consumption analysis of design patterns. In *Proceedings of the International Conference on Machine Learning and Software Engineering*, pages 86–90. Centre for Telematics and Information Technology, University of Twente, 2005.

[30] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th ICSE*, pages 503–514. ACM, 2014.

[31] I. Manotas, C. Sahin, J. Clause, L. Pollock, and K. Winbladh. Investigating the impacts of web servers on web application energy usage. In *GREENS, 2013 2nd International Workshop on*, pages 16–23. IEEE, 2013.

[32] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the sim-panalyzer version 2.0.

[33] A. Noureddine, R. Rouvoy, and L. Seinturier. Unit testing of energy consumption of software libraries. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1200–1205, New York, NY, USA, 2014. ACM.

[34] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about the energy consumption of software? *PeerJ PrePrints*, 3, 2015.

[35] J. J. Park, J. Hong, and S. Lee. Investigation for software power consumption of code refactoring techniques. In *Proc. of the 26th International Conference on Software Engineering and Knowledge (SEKE)*, pages 717–722, 2014.

[36] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2011.

[37] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *ASPLOS 2013*, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM.

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

[39] K. Rasmussen, A. Wilson, and A. Hindle. Green mining: energy consumption of advertisement blocking methods. In H. A. Müller, P. Lago, M. Morisio, N. Meyer, and G. Scanniello, editors, *GREENS 2014*, pages 38–45. ACM, 2014.

[40] 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 *GREENS, 2012*, pages 55–61. IEEE, 2012.

[41] C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *ESEM*, pages 36:1–36:10, New York, NY, USA, 2014. ACM.

[42] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 131–140, Washington, DC, USA, 2014. IEEE Computer Society.

[43] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, pages 1–1, To appear, 2015.

[44] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[45] D. Singh and W. J. Kaiser. The atom leap platform for energy-efficient embedded computing. *Center for Embedded Network Sensing*, 2010.

[46] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[47] Trove. http://trove.starlight-systems.com/.

[48] T.j. watson libraries for analysis (wala). http://wala.sourceforge.net/wiki/index.php/Main_Page.

[49] Watts up. https://www.wattsupmeters.com/secure/products.php?pn=0.

[50] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Aßmann. Energy consumption and efficiency in mobile applications: A user feedback study. In *GreenCom 2013, (iThings/CPSCom) and CPSCom*, pages 134–141. IEEE, 2013.

[51] Xstream. http://xstream.codehaus.org/.

[52] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of DAC*, pages 340–345. ACM, 2000.

[53] C. Zhang, A. Hindle, and D. M. Germán. The impact of user choice on energy consumption. *IEEE Software*, 31(3):69–75, 2014.

[54] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *CODES/ISSS '10*, pages 105–114, New York, NY, USA, 2010. ACM.