Energy Profiles of Java Collections Classes

by

Samir Hasan

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 7, 2016

Keywords: Energy, Java, Collections

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 memory usage and bytecode executed during an operation. Expensive computation tasks in bytecode traces appeared to have 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 a wrong Collections type, as indicated by our profiles, can cost as much as 300% more energy than the more 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 for energy efficiency.

Acknowledgments

I would like to take this opportunity to acknowledge all who helped me in completing my research.

I would like to thank my supervisor Dr. Jeffrey Overbey for his excellent insights, detailed feedback on my work, and the constant support he provided during the difficult phases of my graduate study. I am indebted to my previous supervisor Dr. Munawar Hafiz, who has helped in every phase of the research and academics, and has always made the time to resolve any issues that I had. I highly appreciate his extreme attention to detail, which has shaped my research to what it is today. This would not have been possible without his constant support and dedication.

I am also grateful to my other committee members, Dr. N. Hari Narayanan and Dr. David Umphress, for their support and useful feedback on my research. They have been very encouraging and patient with me.

I would like to express my deepest gratitude to Dr. Abram Hindle and his team at University of Alberta, Canada, for letting me use the GreenMiner device, and making a significant contribution into this work by providing useful suggestions and resources. I am also thankful to Dr. Bram Adams and his team at Polytechnique Montreal, Canada, for sharing their expertise on program analysis.

Finally, I would like to thank my parents who raised me to be able to fulfill my dreams for higher studies, my wife who has always inspired and supported me, and my friends who believe in me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Energy consumption is rapidly becoming an emerging topic for software engineering and research [8,9,23,29,31,36,40,42,43]. In particular, the scale of data centers and limited battery lifetime of ubiquitous mobile devices have forced the owners 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 [35,51]. If a guideline was available, the developers could make informed choices in building "greener" systems by choosing the most suitable energy-efficient coding alternative.

1.1 Energy Concerns in Software

A software system during execution makes use of hardware resources like CPU, memory, hard disk, network, and display—all of which are electrical components that require some form of energy to function. Given multiple executing applications, cloud computing, and virtual machines, controlling energy at the hardware device level alone is not sufficient. Hence, researchers have recently started focusing on autotuning the energy consumption inside software to optimize energy-efficiency [8,9,14,15,31]. Götz and colleagues [14,15] contributed the initial work following the autotuning optimization approaches in performance improvement (e.g., [38]). 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 [31] 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, 30, 41] and refactoring [36, 42]. However, they were not able to provide specific guidelines, perhaps because the energy footprints of these coding decisions were too small. Manotas and colleagues [31] 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.

1.2 Java Collections Classes

Java Collections classes store group of objects and provide API to access, modify, or iterate over the elements. Java ships with the Java Collections Framework (JCF) [25] that provides reusable and convenient implementations of popular data structures and algorithms. There are also many third-party implementations of similar structures.

In this work, we also studied two third-party implementations: Apache Commons Collections (ACC) [2] and Trove [48]. Form ACC, we studied implementations that are alternative to those already in JCF. The Trove collections hold only primitive data types; the goal is to reduce memory usage and improve performance (Trove requires three times less heap space than JCF implementations for larger collections [48]). The Collections classes we studied are shown in Table 1.1.

The List interface defines an API to insert items at the end and at an any particular index. Other common operations include iterating the list and retrieving an element from the list with an index. We studied five list implementations from the three frameworks. Although they provide the same API, the underlying data layout are different, accounting for different execution performance of the lists. For example, TreeList allows fast ($O(\lg n)$) insertions at a random index in the list, while it is a $O(n)$ operation on a LinkedList.

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

Table 1.1: Profiled Collections Classes

We studied six alternative Map implementations from the three frameworks. Common operations on map include inserting a key-value pair into the map, iterating over the entries and looking for a particular key in the map to retrieve the corresponding value. These maps also employ different data structures internally.

Similarly, we profiled six Set implementations for energy consumption. Sets store unique elements and provides API to iterate over the elements or perform fast lookups. HashSet, TreeSet, LinkedHashSet and MapBackedSet actually use a map internally to emulate the set behavior. TIntHashSet and ListOrderedSet use a different technique to manage the data. We looked into how the energy consumption of these implementations compare.

1.3 Energy Profiles

Alternative Collections classes have common API but perform differently in terms of execution time and energy consumption, among other factors. Energy profiles illustrate how much energy is consumed when a particular API is invoked for a varying number of times on a specific Collections instance. These are graphs plotted with the frequency of invocation on the x-axis and energy consumed (joules) on the y-axis.

We created energy consumption profiles of commonly used API methods for variants of three Collections datatypes: List, Map, and Set. 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 programmers 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 [31].

1.4 Thesis Statement and Contributions

Collections in Java provide some of the building blocks used by developers. When writing programs, developers are typically unaware of the energy consequences of using these blocks. If energy profiles of the alternative coding idioms, such as the Collections implementations, are available, developers can use them as a guideline to choose the "green" option based on the coding context.

Our work is a case study that explores the energy footprints of different API from some of the Collections implementations. We make 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 (Chapters 4).
- We explored two possible alternatives to explain energy consumption differences between operations (Chapter 5).
- We evaluate on real applications that the alternative Collections classes can be swapped to predictably improve or worsen energy consumption (Chapter 6).

1.5 Thesis Outline

The thesis is outlined as follows:

- Chapter 2 discusses related work.
- Chapter 3 introduces GreenMiner, the energy measurement infrastructure that we used perform our energy analysis, and describes our experimental setup for obtaining the energy profiles.
- Chapter 4 demonstrates the energy profiles of the Collections implementations, the impact that the number and type of data elements have on the overall energy consumption, and a guideline for developers towards building "green" systems.
- Chapter 5 addresses the factors that may explain the energy profiles we illustrated in Chapter 4.
- Chapter 6 discusses the results of evaluating the profiles in real Java applicaitons and libraries.
- Chapter 7 provides concluding remarks and suggests future work.

Chapter 2

Related Work

Energy consumption of software has earned the interest of many researchers in recent years. The motivation for research in this area 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 the energy bottlenecks are, and which programming practices should be avoided [35,39,51,54]. Pinto et. al. [39] explored techniques for writing energy efficient code and suggested several strategies, such as doing minimum I/O, performing operations in bulks, minimizing interaction with the hardware, making use of concurrent programming, using efficient data structures and performing lazy initialization when possible. Researchers have been trying to relate code change with the energy consumption by employing a variety of tools and techniques for energy measurement or estimation. Several auto-tuning approaches have been explored to assist developers in making energy-efficient coding choices. The next few sections highlight the significant findings in these areas.

2.1 Energy Impact of Code Change

There has been a large body of empirical work measuring the impact of code change in various domains. Sahin et. al. [42] reported that code refactorings can have an impact on the application's energy consumption, and common factors such as execution time and dynamic execution counts are not enough to predict the energy usage. In a related study, Park et. al. [36] studied the impact of the 63 refactoring techniques suggested by Martin Fowler [13] on energy consumption. They shortlisted 33 techniques that are actually energy-efficient. Design patterns also have an impact as suggested by Bunse and Stiemer [10]. They found that the decorator pattern consumes more energy and observed that employing patterns may not always be a good idea as far as energy-efficiency is concerned. Litke et. al. [30] studied the impact of Factory Method, Adapter, Observer, Bridge and Composite pattern, and found that only the first three have noticeable energy impact. Moreover, these patterns did not cause a significant increase in the energy usage. Sahin et. al. [41] conducted a similar study and concluded that design patterns can cause the energy consumption to both increase and decrease, and it is unlikely that the energy usage can be predicted based solely on the employed patterns. Abtahizadeh et. al. explored design patterns applied to recurrent problems in the cloud, and found that some patterns are effective in reducing energy consumption of a system [1]. A common technique to prevent piracy is obfuscating the code. The impact of code obfuscation have been found to be statistically significant, although for mobile applications the real magnitude may not be noticeable [43].

Hindle proposed Green Mining to study how changes across software versions affect energy consumption [21]. The paper compares various branches and versions from Firefox and Azeurus/Vuze, examines the source-level changes that occur with time and attempt to relate software metrics to energy consumption. Sorting algorithms have also been found to have an impact on the energy consumption of a program [8, 9]. Different sorting techniques have different energy consumptions, and there seemed to be no direct correlation between the energy consumed and the runtime complexity. Hunt et. al. studied the energy consumption of lock-free data structures [23]. The results show that the lock-free data structures performed better and consumed less energy than their locking counterparts.

Manotas et. al. performed an empirical study on the energy consumption of web applications [32]. They profiled different web servers and found that web servers make a significant contribution in the energy usage. Moreover, different web servers contribute in different amounts, so the choice can be directed by energy constraints.

Researchers have also looked at energy usage of various aspects of Android applications. Linares et. al. performed an empirical study to investigate the API calls in the Android development framework that consume a significant amount of energy [29]. By studying the data from 55 free Android applications, the authors found that some design choices, such as implementing an MVC or a persistence layer with a relational database, can contribute significantly to the overall energy consumption of an application. Saborido et. al. proposed a recommendation system that help users to pick the most energy-efficient as well as highly-rated application for a particular category, such as email, browser, camera, etc [24]. They achieved energy savings of 16.61% when following the recommendations. Advertisements and ad-blocking have been found to have a negative effect on the energy consumption of applications [17, 40].

Li et al. [28] repeatedly profiled Java bytecode instructions to link source code and bytecode to energy consumption in order to estimate the energy usage of a line of Java code. JalenUnit [34] uses PowerAPI and statistical execution sampling to automatically generate benchmarks to measure the energy consumption of an API.

2.2 Measurement Techniques for Energy

Researchers have measured energy in a number of ways. Hardware systems such as the Atom LEAP platform [46] and WattsUp meters [50] can measure actual power consumed by an application. Cycle accurate simulators such as SoftWatt [19], Sim-Panalyzer [33], and simplepower [53] provide an energy estimate by simulating CPU cycles for each component used in executing the application. Estimation based approaches [5–7, 12, 20, 34, 47] use empirical data to propose a model for estimating energy consumption. Pathak et al. [37] 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. [18] fingerprinted modules for their energy consumption profile. Zhang et al. [55] describe an online profiler called PowerTutor that models energy consumption by aggregate models of individual components such as network and CPU.

2.3 Auto-tuning for Energy Efficiency

Our work is closest to the approach taken by Manotas et al. [31]. 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. In this empirical study, the authors investigated 7 applications and demonstrated that the automated system can be used to improve the energy consumption of an application, although it suffers from hours of processing time. 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. This analysis also completes very fast, usually in a minute.

Another closely related work is Chameleon [45], 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.

Chapter 3

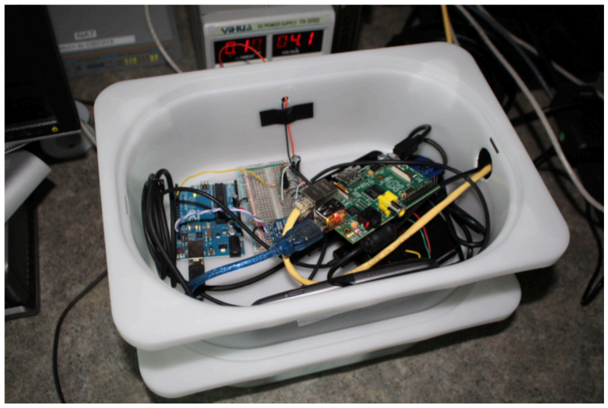
Energy Measurement Infrastructure Setup

We used GreenMiner's hardware infrastructure to measure the actual energy consumed by our test programs [22]. GreenMiner (Figure 3.1) 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. GreenMiner is developed and maintained by a team of researchers at University of Alberta, Canada.

3.1 GreenMiner Hardware

The main components of the GreenMiner infrastructure are a Raspberry Pi, an Android test device, the Adafruit INA219 IC and an Arduino Uno micro-controller as shown in Figure 3.1. The Raspberry Pi test-bed, the GreenMiner client, runs GNU/Linux equipped with utilities such as the ADB (Android Debug Bridge), python and bash. It uses 8GB SD cards for storage and filesystem. The Pi provides an serial-USB interface through which it controls the Android test device.

The Samsung Galaxy Nexus phone running Android OS 4.2.2 was chosen as the test device for GreenMiner. The Pi controls it through ADB, executing android programs/tests on the device while the Arduino Uno monitors its energy consumption during the run. The INA219 chip is connected to the power supply of the phone and measures voltage, amperage, and wattage (W) consumed by the phone. The Arduino reads off these measurements. When the test application finishes running on the Android device, the Raspberry Pi gathers power and energy consumption data from the Arduino, compiles them into a report with meta-data, and uploads them to GreenMiner webservice.


(a) GreenMiner hardware (all parts)

(b) A Galaxy Nexus test device

Figure 3.1: The GreenMiner infrastructure [22].

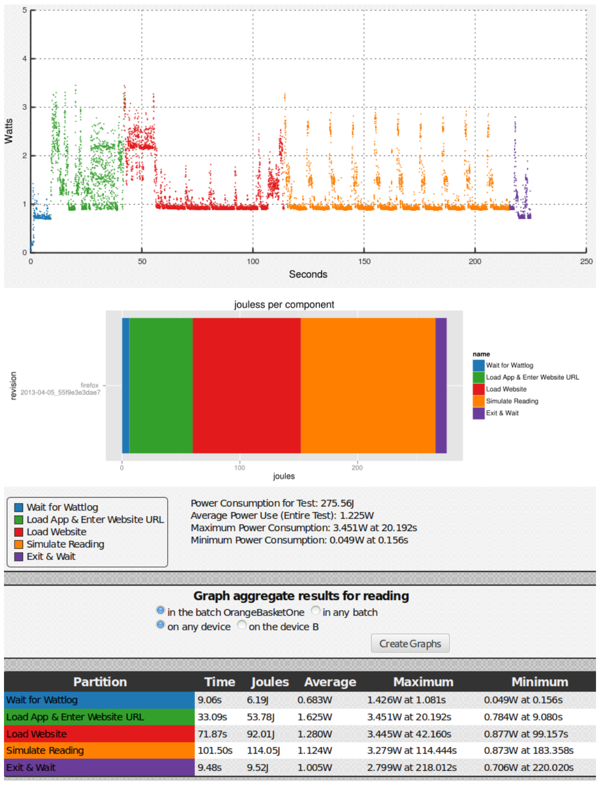
3.2 GreenMiner Webservice

The GreenMiner webservice provides three primary services: distribution of test data, visualization of the data, and controlling GreenMiner to schedule and monitor tests on the system remotely.

3.2.1 Distribution and Visualization of Test Data

Once data is collected on the GreenMiner client Raspberry Pi, it is made available at the GreenMiner's website. A user visiting the site can choose one of the tests from a list of previous runs and discover details on the test run. Figure 3.2 shows an example. The power plot shows wattage of the Android device against time. Different colors on the plot represent the different tasks as outlined in the legend and stacked plot. The page also provides energy measurements in Joules, power usage, execution time, and other statistics on the test. The download button at the top-right corner of page allows the user to obtain the experiment data that can be used to prepare other customized graphs and perform further analysis.

A powerful feature of the GreenMiner website is its ability to aggregate results from multiple test runs and prepare common visualizations useful for studying performance data. Graphs such as T-test similarity matrices showing statistical significance between different



🟦 Wait for Wattlog
🟩 Load App & Enter Website URL
🟥 Load Website
🟧 Simulate Reading
🟪 Exit & Wait

Power Consumption for Test: 275.56J
Average Power Use (Entire Test): 1.225W
Maximum Power Consumption: 3.451W at 20.192s
Minimum Power Consumption: 0.049W at 0.156s

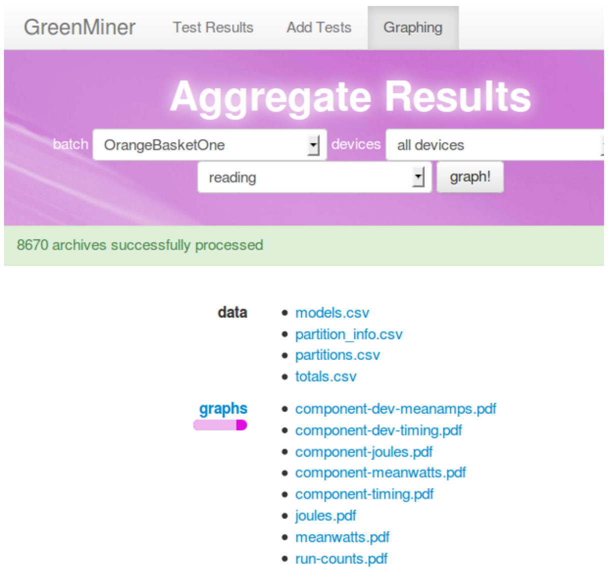
Graph aggregate results for reading

[✓ in the batch OrangeBasketOne] in any batch
[✓ on any device] on the device B

Create Graphs

Partition	Time	Joules	Average	Maximum	Minimum
Wait for Wattlog	9.06s	6.19J	0.683W	1.426W at 1.081s	0.049W at 0.156s
Load App & Enter Website URL	33.09s	53.78J	1.625W	3.451W at 20.192s	0.784W at 9.080s
Load Website	71.87s	92.01J	1.280W	3.445W at 42.160s	0.877W at 99.157s
Simulate Reading	101.50s	114.05J	1.124W	3.279W at 114.444s	0.873W at 183.358s
Exit & Wait	9.48s	9.52J	1.005W	2.799W at 218.012s	0.706W at 220.020s

(a) Individual test report.



Aggregate Results

batch OrangeBasketOne devices all devices
reading graph!

8670 archives successfully processed

data
- models.csv
- partition_info.csv
- partitions.csv
- totals.csv

graphs
- component-dev-meanamps.pdf
- component-dev-timing.pdf
- component-joules.pdf
- component-meanwatts.pdf
- component-timing.pdf
- joules.pdf
- meanwatts.pdf
- run-counts.pdf

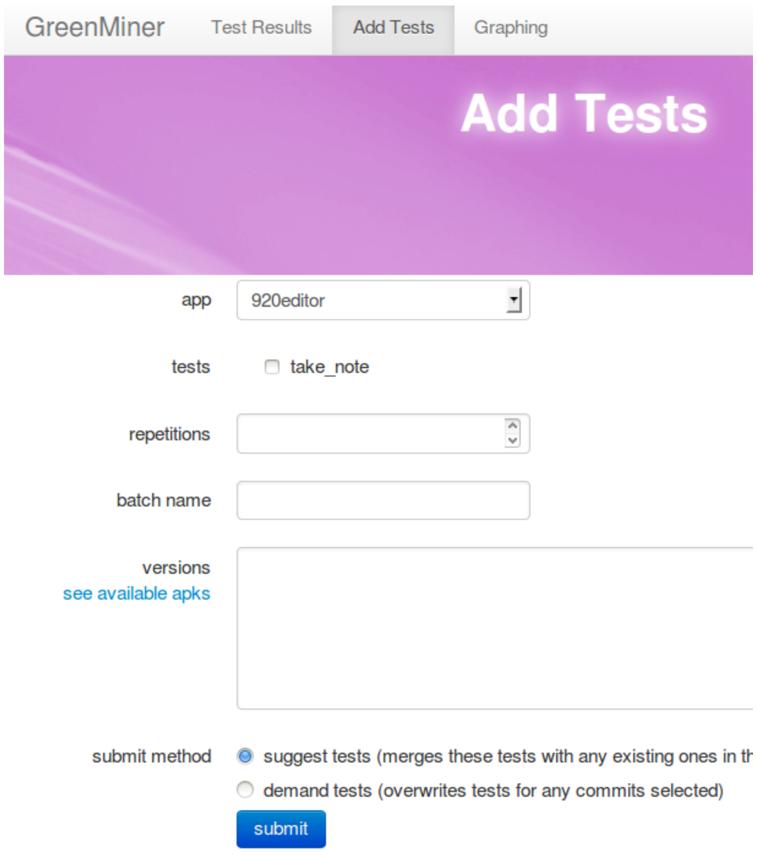
(b) Vizualizations on test data for a single test.

Figure 3.2: The GreenMiner test report [22].

tests, stacked box-plots, run count plots and per-device graphs are some of the frequently used visualizations to facilitate energy consumption analysis in GreenMiner.

3.2.2 Remote Test Management

The GreenMiner website provides interfaces to view currently running tests, schedule new runs and stop running tests. It also shows logging information generated at the Pi machines when the tests run on the Android device. This is very helpful since users can get the log report when tests break at the remote end. Figure 3.3 shows the interface of scheduling a test run into the queue.



app: 920editor

tests: ☐ take_note

repetitions:

batch name:

versions:

submit method: ● suggest tests (merges these tests with any existing ones in th

○ demand tests (overwrites tests for any commits selected)

Figure 3.3: A GreenMiner interface for scheduling a test run [22].

3.3 Experimental Setup

The previous section introduced GreenMiner as an energy consumption monitor for an Android device. This is convenient since we wanted to profile Java Collections implementations and Android applications are built with Java. When we run an application or a test suite on the Android test device, GreenMiner captures the energy footprints at intervals and prepares a report. There were several factors that we needed to account for the measurements, as discussed in subsequent sections. After collecting the data, we cleaned up and aggregated them with scripts and prepared them for graphing with R. A workflow of our measurement process is illustrated in Figure 3.4.

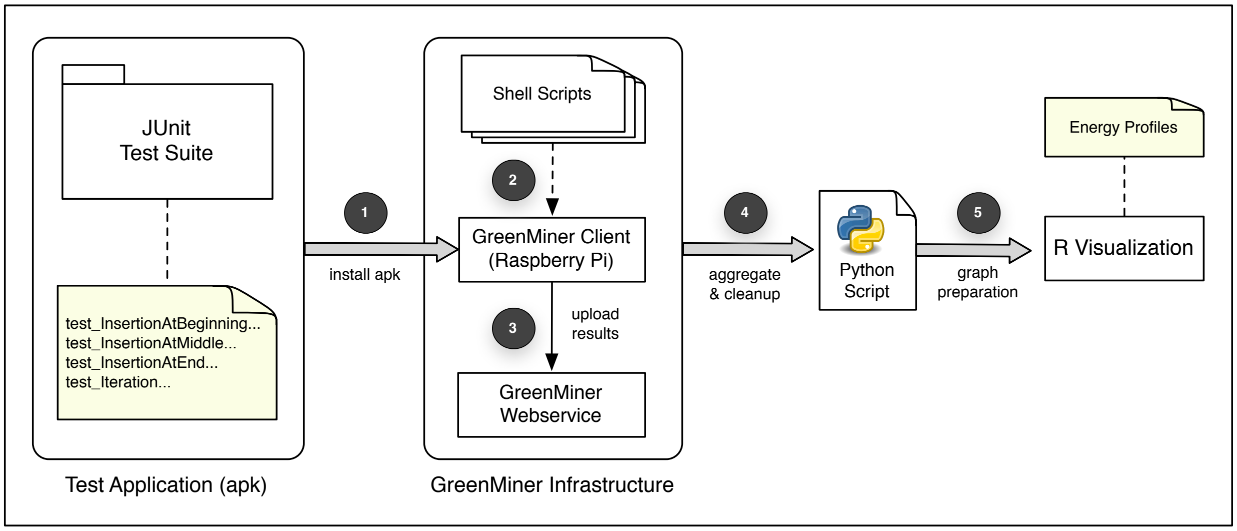


Figure 3.4: Experimentation workflow.

3.3.1 Test Design

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. We measured and recorded the energy consumed in joules (J) by the test with GreenMiner (Figure 3.4).

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 1.1). 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).

3.3.2 Measurements

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

```
public void setUp() throws Exception  {
    arrayList = new ArrayList<Integer>(SIZE);
    tIntArrayList = new TIntArrayList(SIZE);
    linkedList = new LinkedList<Integer>();
    treeList = new TreeList<Integer>();
    tIntLinkedList = new TIntLinkedList();
}
```

Figure 3.5: The `setUp()` method for inserting items into List instances.

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 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.

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 as shown in Figure 3.5.

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()` as shown in Figure 3.6. All the unit tests were designed similarly.

```
public void test_InsertionAtEndOfLinkedList() throws
    Exception {

  // Parameters.REPS is 20
  for(int rep = 0; rep < Parameters.REPS; ++rep) {
    setUp();
    for(int i = 0; i < SIZE; ++i)  // SIZE is 50
      this.linkedList.add(i);
    tearDown();
  }
}
```

Figure 3.6: Code for inserting items at the end of a LinkedList instance.

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.

Chapter 4

Results

We profiled the energy consumption of some of the common API methods provided by List, Map, and Set implementations, and record how it varies with input sizes. Using these profiles, we tried to identify which List, Map and Set implementations are the most energy efficient for each of the common operations: insertion, iteration and random access. This led us to produce a general guideline for the developers to help them choose the right implementation based on the different usage scenarios collectively. We also studied whether other variables have an impact on the energy consumption of the collections, such as the element size and the number of elements in the list. The next few sections discuss our findings in detail.

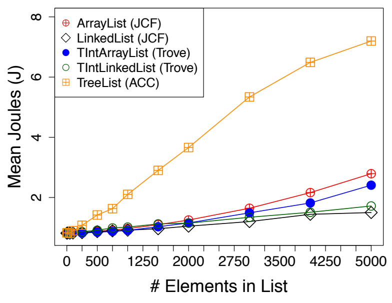
4.1 List

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 end, Trove's ArrayList is the most energy efficient, followed by JCF's ArrayList.

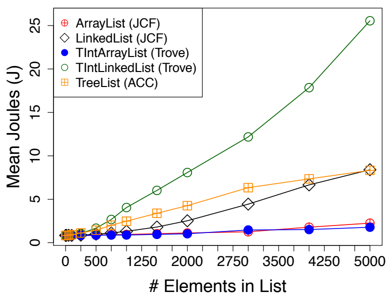
Figure 4.1(a), 4.1(b), and 4.1(c) demonstrate the energy consumption trends for insertion tests for the five List implementations.

Insertion

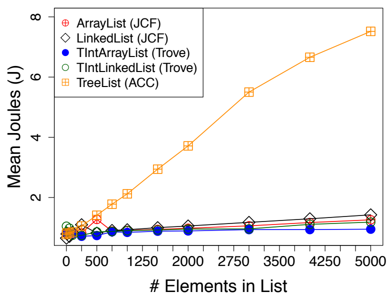
For small sizes (1–500), the difference in energy consumption for insertions at the beginning of the list is not evident for all but two implementations—TreeList and TIntArrayList. Even at size 250, TreeList consumes $\approx 31\%$ more energy than TIntArrayList, while others



(a) Insertion at beginning



(b) Insertion in middle



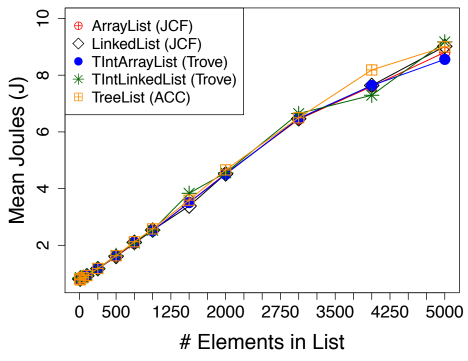
(c) Insertion at end

Figure 4.1: List: Energy profiles for insertions. Figures (a) – (c) shows list instances that have been initialized with capacity if the API allows it (eg. ArrayList).

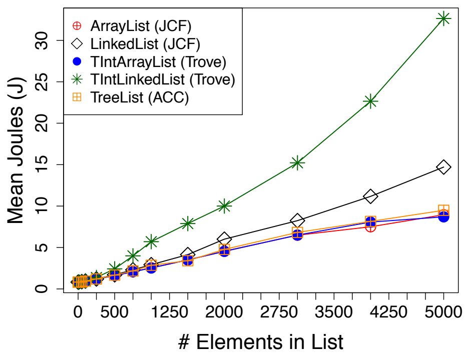
consume about the same energy. However, for larger sizes, LinkedList begins to perform much better than others. At input size of 5,000, LinkedList consumes ≈ 13% less energy than TIntLinkedList, the next best performer. Compared to the worst performing TreeList, LinkedList consumes ≈ 79% less energy.

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 which 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 1000 for all lists, with the exception of TreeList, which has a noticeable degradation of ≈ 32% at the size 250 (Figure 4.1(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.



(a) Iteration



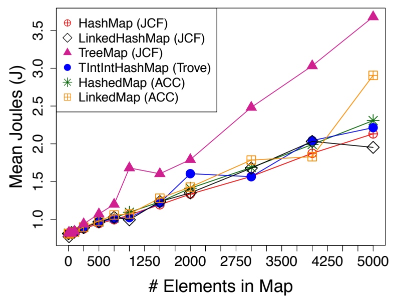
(b) Random access

Figure 4.2: Iteration and random access performance of List implementaitons.

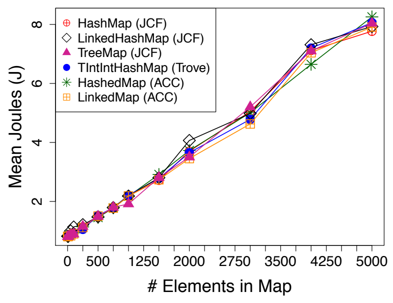
We also gathered similar profiles for the case when ArrayList and TIntArrayList are not set to a predefined capacity during creation. Uninitialized array lists need to reallocate memory when current capacity is not sufficient. Due to this dynamic resizing, we expected a difference in energy consumption 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

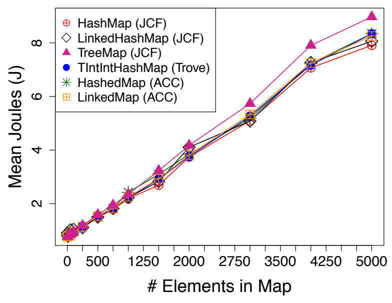
Figure 4.2(a) shows the energy consumption profile for iteration with an iterator. For small sizes, iterating an ArrayList is slightly more energy efficient, while for 5000 items, it has a maximum energy savings of ≈ 4%. The results show that there is not much difference when comparing energy consumption of iteration over the lists.



(a) Insertion



(b) Iteration



(c) Random Query

Figure 4.3: Energy profiles for insertions, iteration, and queries on random keys on Map

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 4.2(b). 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.

4.2 Map

Key Result: HashMap is the most energy efficient alternative for insertions and random query. If insertion order is required to 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 4.3(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 5000 insertions, it is ≈ 73% more expensive than HashMap and ≈ 88% more expensive than LinkedHashMap.

All other maps perform equally well for sizes up to 1000. Interestingly, HashMap performs consistently better than all other maps until size 5000 where LinkedHashMap has a drop in energy consumption and saves $\approx 8\%$ energy over HashMap. For most of the cases, Trove's TIntIntHashMap uses slightly higher energies than HashMap. This was surprising, since Trove implementation with primitive data types did not improve upon JCF HashMap, and for some input sizes it was noticeably worse.

Iteration

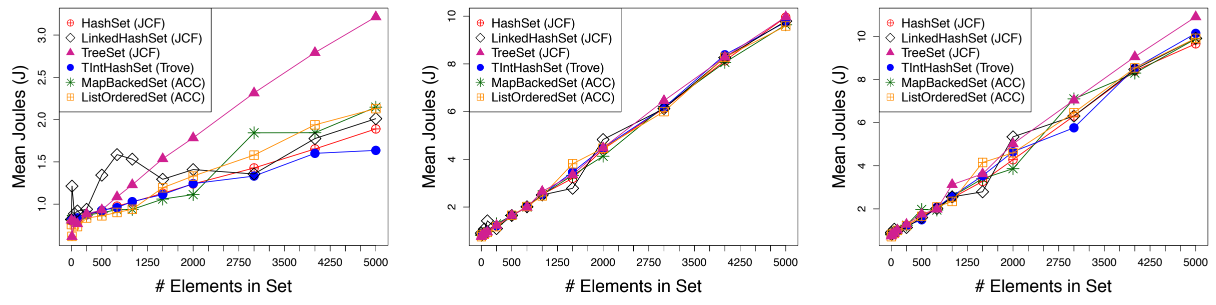
Similar to our findings for lists, the iteration performance is almost the same for all implementations (Figure 4.3(b)). For larger lists, JCF's HashMap requires a little less energy, while ACC's HashedMap ended up being the most expensive. However, the differences are very small for large lists, and even more so than for smaller ones.

Query

The random query performance has an interesting trend as shown in Figure 4.3(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.

4.3 Set

Key Result: HashSet is the most energy efficient alternative for insertions and random query. 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.



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

Figure 4.4: Energy profiles for insertions, iteration, and queries on random keys on Set.

Insertion

Figure 4.4(a) shows the energy consumed by inserting values into Set implementations. 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 ≈ 13% energy over HashSet and ≈ 49% over TreeSet.

Iteration

The iteration performance, as shown in Figure 4.4(b), is very similar for all implementations—there are no apparent differences for sizes up to 1000. At the larger sizes, ACC's ListOrderedSet is the most energy efficient, with a maximum energy saving of ≈ 4%. Again, there are often larger energy savings between smaller input sizes than there are between larger; for example, at size 1500 ACC's ListOrderedSet has an ≈ 27% savings over JCF's LinkedHashSet where the savings between the same implementations at size 5000 is only a mere ≈ 2% savings.

Query

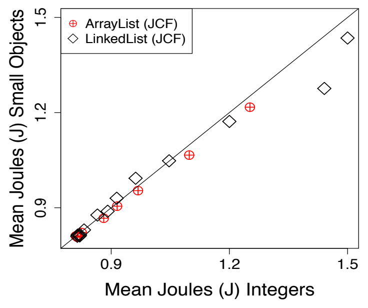
The random query performance is shown in Figure 4.4(c). There are many energy spikes throughout the various input sizes. Interestingly, the largest differences between implementations are in the medium size inputs, between sizes 1000 and 3000. Another notable trend is that TreeMap starts out in the smaller inputs to be one of the most efficient implementations for smaller sizes and then for larger sizes has an $\approx 13\%$ degradation from the optimal performing HashSet. Whereas at size 50, TreeSet actually saves $\approx 4\%$ over HashSet.

4.4 Impact of Data Type and Collection Size

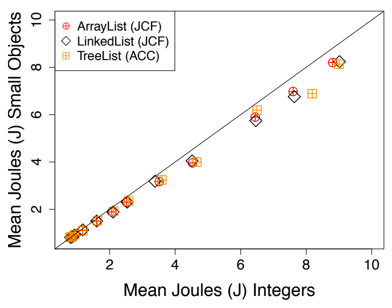
Does different data types account for different energy consumptions of the Collections types? To answer this question, we ran another set of experiments on the Collections instances as we did for preparing the energy profiles, but changed the the elements to small sized objects instead of `Integer` instances. We compared these new profiles against the previous set and discuss the differences. We also looked into our energy profiles to see if the difference in energy consumption between alternative Collections types is significant for small versus large lists.

4.4.1 Data Type of Elements

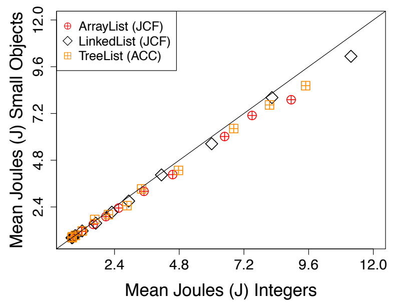
We report here the tests on inserting into List instances of small objects. We expected the energy consumption to be higher than when running the tests on List of integers due to storing larger data inside the lists. But, the energy profiles of integer runs were consistently more than those of the small objects. Figure 4.5 shows a scatterplot comparing the energy consequences of List of integers vs List of small objects; the results are shown for ArrayList and LinkedList. Some inputs of the ArrayList are not shown since it skews the graph. The majority of the points lie below the line $y = x$, whereas we were expecting for all points to lie above it. For size 4000, inserting an integer in the beginning of a JCF LinkedList is $\approx 13\%$ more expensive than inserting a small object. Though only insertion at beginning of LinkedList and ArrayList are shown, these results are consistent with other scenarios. More details and graphs are on the project webpage.



(a) Insertion



(b) Iteration



(c) Random Query

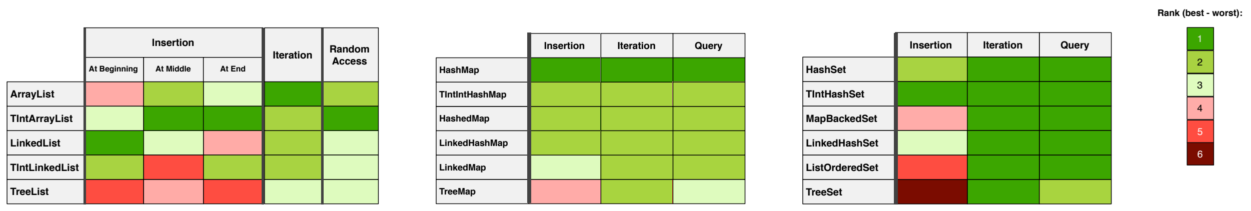
Figure 4.5: Comparison of energies for operations on lists of integers versus small objects.

When running List insertion and iteration tests on small objects, we expected the energy consumption to be higher than the integer based results. However, the results show that integers are actually as energy expensive, and in many cases more expensive, than small objects as elements. 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 are constantly adding and shedding this Integer wrapper at will. The results are likely due to this.

4.4.2 Size of Collections

From what input size on the energy differences become significant? To determine the statistical significance of energy differences, we computed the 95% confidence intervals of the energy consumption measurements for each alternative collection and each size. For a particular size, we compared the confidence intervals 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 collections in a similar way, we found



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

Figure 4.6: Color map showing the rank of each List, Map and Set implementations based on the use case. Green identifies an implementation as energy efficient, while red denotes it as energy hungry. Therefore, the greener the color, the better.

500 to be an appropriate threshold across all Collections types. Therefore, for input sizes 1–500, all alternative implementations of List, Map, and Set perform equally well. The differences become significant when there are more than 500 elements in the collection.

4.5 A Guideline for Developers

The previous two sections describe the energy consumption trends of the Collections instances for insertion, iteration and query APIs. Equipped with this information, can developers choose the most energy efficient implementation wisely based on the usage scenarios? We summarized our results to prepare a guideline that may help developers in making such choices.

In Section 4.4.2, we saw that above the minimum threshold on the input size of 500 items, we can use our profiles to choose the most energy efficient implementation based on the way the Collections classes are used. For example, TIntArrayList is the most energy efficient list implementation, followed by ArrayList. For maps, HashMap performs the best. For Sets, HashSet is the most energy efficient with TIntHashSet as a close second. Figure 4.6 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 six. A row in the table with more green in it is likely to be energy efficient on average.

For example, if a developer is looking for a List implementation that will be used heavily for insertions both at the beginning and at the end but not at the middle, his best pick would be TIntArrayList (if applicable) as it is the greenest in these columns, followed by LinkedList, then ArrayList, and so on. For iteration and random access, either TIntArrayList or ArrayList works fine, since both of them are equally green when both the columns are considered together. HashMap is excellent for all operations we studied. All tree-based Collections implementations seem to perform poorly on insertions compared to their alternatives. However, if the sorted order of the elements needs to be preserved, there may not be any better choice available.

In general, TIntArrayList, HashMap and HashSet are the standout Collections implementations, followed closely by ArrayList and TIntHashSet. Therefore, lists stored as arrays are usually preferred. 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.

We will demonstrate the application of these color maps in Chapter 6, where we showed that the energy consumption of real application and libraries can be improved or degraded by following the trends depicted in the maps. The trends shown here may educate the developers and help them choose the right Collections type ahead of time during their coding efforts.

Chapter 5

Why these Energy Differences?

We carried out further investigations to discover the key factors that may explain the different energy consumption 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 insertions into the Collections, since energy differences are significant when inserting items into Collections instances.

5.1 Memory Usage

We recorded memory consumption for List, Map and Set instances before and after invoking the `add()` or `put(...)` operation, while adding 500 items to the collection. We chose 500, since we showed in Section 4.4.2 that Collections with 500 items or more show statistically significant differences in energy consumption. We wrote new tests using the `java.lang.Runtime` class to track the memory consumed before and after invoking the `add()` or `put(...)` operation, as shown in Figure 5.1.

Figure 5.2(a) shows the resulting memory consumption 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. Since we did find different energy profiles (Figures 4.1(a)–4.1(c)) for the different insertion approaches, this suggests that memory consumption is not a (significant) driving factor behind the energy consumption in this context.

For Map and Set implementations, we again found similar cases where the trend in memory usage contradicts with that in energy consumption. For example, TreeMap and

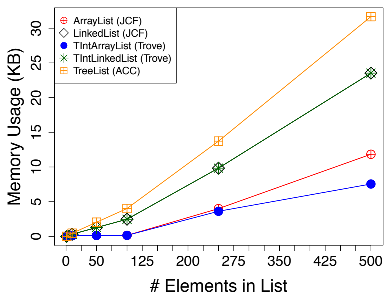
```
...
static HashSet<Integer> set = new HashSet<Integer>();
static ArrayList<Long> usage = new ArrayList<Long>(20);
static {
  set.addAll(Arrays.asList(new Integer[]{0, 1, 10, 50, 100,
     250, 500,
        750, 1000, 1500, 2000, 3000, 4000, 5000}));
}

public static void recordMemory(int iteration) {
  if(set.contains(iteration+1)) {
    Runtime rt = Runtime.getRuntime();
    usage.add(rt.totalMemory() - rt.freeMemory());
 }
}

public void test_InsertionAtBeginningOfArrayList()
throws Exception {

  recordMemory(-1);
  this.arrayList = new ArrayList<Integer>();
  for(int i = 0; i < SIZE; ++i) {
    this.arrayList.add(0, i);
    recordMemory(i);
  }
  writeMemoryInfo("ins_at_beg_AL");
}
...
```

Figure 5.1: Code for measuring the memory consumption for insertions at the beginning of an ArrayList instance.

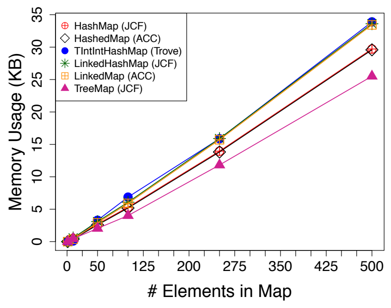
TreeSet use the least amount of memory among their respective alternatives (Figures 5.2(b) and 5.2(c)), but they are both the most energy hungry implementations (Figures 4.3(a) and 4.4(a)). Apparently, there seems to be no direct relationship between the memory usage and energy consumption even for maps and sets.

5.2 Executed Dalvik Bytecodes

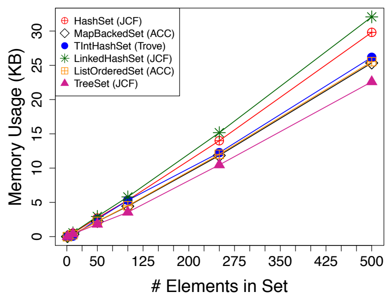
We generated bytecode traces during the lifecycle of an `add()` operation of two List instances (ArrayList and LinkedList) and compared them to reason about their energy differences. We left further analysis of other Collections type as a future work.



(a) Insertions in List instances.



(b) Insertions in Map instances.



(c) Insertions in Set instances.

Figure 5.2: Memory usage of List, Map and Set instances during insertions.

To identify which parts of an Android application's bytecode are being executed, we first use dexdump to extract the application's bytecode from its dex file (which is included in the apk package), then instrument the bytecode using the AndBug debugger tool, which implements the Java Debug Wire Protocol (JDWP). In particular, this tool allows to put breakpoints on any source code line of interest, and, upon reaching such a breakpoint during execution, prints out the corresponding source code line's identifier. Since we are interested in knowing all bytecode lines that are being executed, we added breakpoints on each source code line. We then ran the android App using AndBug, generating the execution traces. Finally, to determine which bytecode instructions have been executed (since one source code line maps to one or more bytecode instructions), we use the bytecode file's internal mapping from bytecode identifiers to the line numbers of the corresponding source code.

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 as shown in Figure 5.3. 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 500th element is inserted to a list, `iget-object` is executed 63 times more than the next frequently occurring instruction.

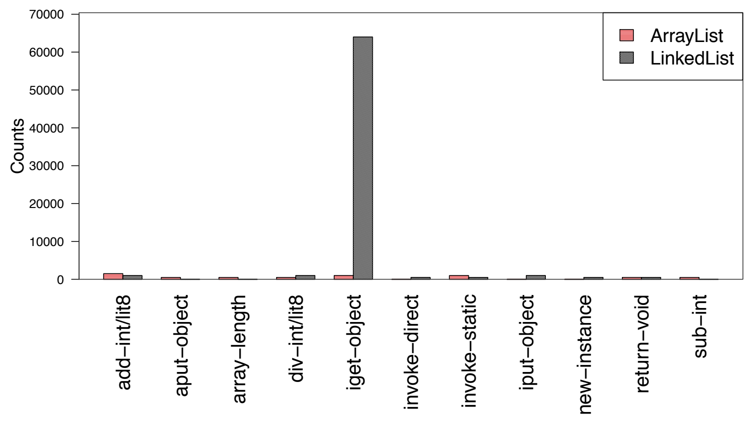
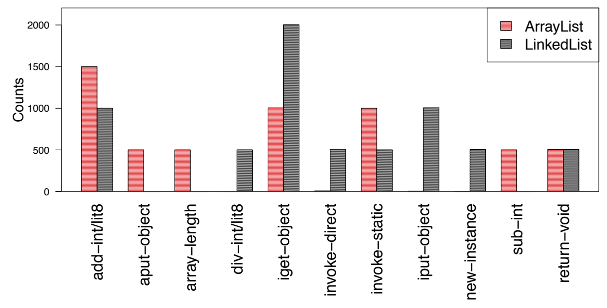


Figure 5.3: Frequently executed bytecodes during insertions at the middle on ArrayList and LinkedList instances.

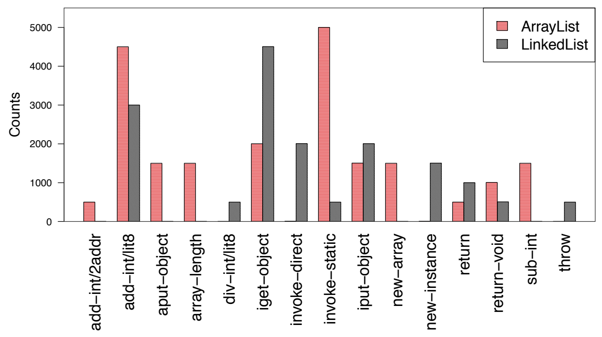
This may explain why LinkedList consumes more energy than ArrayList, as shown in the energy profile (Figure 4.1(b)).

The impact of bytecodes are less obvious when elements are inserted at the beginning or at the end as shown (Figures 5.4(a) and 5.4(b)). When inserting at the beginning, `invoke-static` dominates the execution for an ArrayList (used to execute the expensive `System.arraycopy()` method). For every invocation of `add()`, LinkedList executes 2 extra `iget-object` instruction while ArrayList executes 2 `invoke-static` instruction. The `iget-object` instruction in LinkedList loads a value into a register. The `invoke-static` for ArrayList executes the `System.arraycopy()` method. Apparently, ArrayList is performing more extensive work than LinkedList with respect to these two bytecodes. 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.

The bytecodes for end-insertions present a more complex scenario. There are more `invoke-static` operations for ArrayList (10 times) than there are `iget-object` for LinkedList (9 times). If these were the only 'important' instructions, ArrayList would probably have consumed more energy than LinkedList. However, unlike the case with insertions at the beginning, we have other instructions that we also need to consider: `iput-object` and `invoke-direct`. Since now there are too many variables, and we do not know the relative



(a) Insertions at the beginning.



(b) Insertions at the end.

Figure 5.4: Frequently executed bytecodes for insertion at the beginning and end of ArrayList and LinkedList instances.

weights of each on the runtime performance, we can not directly deduce a bytecode execution pattern that explains why ArrayList consumes less energy than LinkedList for end-insertions. Hence, this analysis is not enough to explain why ArrayList performs better in this context. We shall address this as future work.

Chapter 6

Evaluation

The energy profiles compare the Collections classes for each API method and suggest better alternatives. 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 the following research questions:

RQ1. Do the different Collections classes have energy impact in real applications compared to what we found for similar collections in the profiles? How big is the impact?

RQ2. Can we use the energy profiles to switch to an alternative collection and improve (or degrade) the energy consumption of an application?

To answer RQ1, 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 [31]. 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 (RQ2): we create "good" and "bad" versions of the original program when possible, and compare their energy consumption using GreenMiner.

Using the methodology of Figure 6.1, we studied the energy consumption of four popular Java libraries (Google Gson [16], Apache Commons Math [4], XStream [52], Apache Commons Configuration [3]), an open source email client (K-9 Mail [27]), and a Stock Exchange Trading Simulator application. Each library came with a large test suite (16–137 KLOC). We analyzed the list datatypes used in the code to create usage profiles, i.e., which

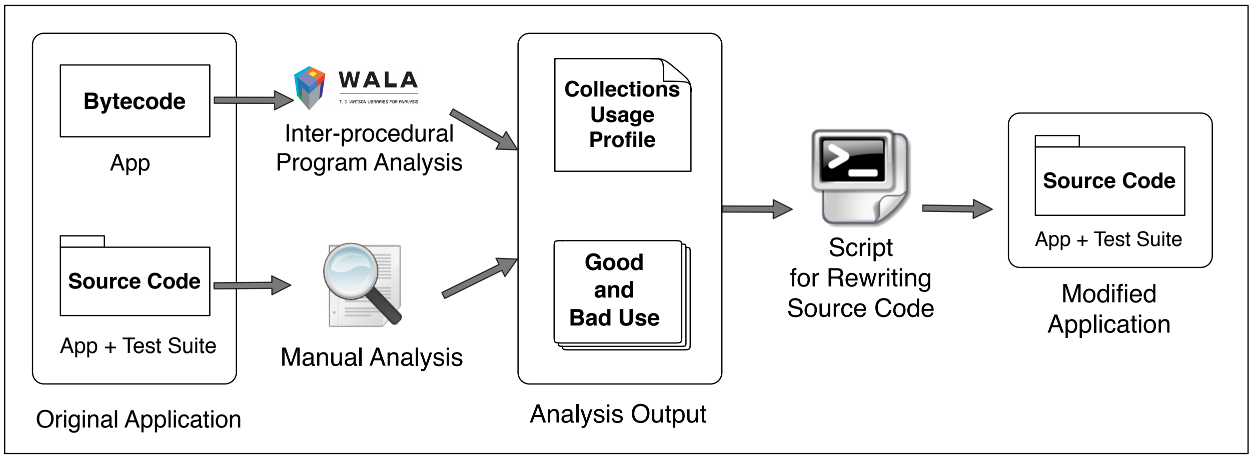


Figure 6.1: Evaluation Workflow

API methods are being invoked and where. We wrote an inter-procedural program analyzer based on WALA [49] 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 it is passed as a return type. 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 instances of Collections passed as parameters 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 Section 4.5. When creating a "good" version, we looked into the usage of Collections instances to see whether swapping one with another 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 insertions at the end and iterations, our WALA program will detect it and indicate that we can improve energy consumption by replacing LinkedList with ArrayList.

Similarly, we prepared the "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, the '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. [31]). Furthermore, we chose to deal only with lists during our evaluation. There are two reasons behind the 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

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 6.1: Libraries and applications for evaluating the energy profiles

LinkedList. In K-9 Mail, we had comparatively more LinkedLists, and thereby had a scope to show improvements. We therefore had both a “good” and a “bad” version for it. We also made “good” changes in the Apache Commons Configuration library and the Stock Exchange Trading Simulator application, since they also use LinkedLists for operations that could be optimized for energy consumption.

The next sections describe the results of our analyses for the six applications. To address RQ1, we report the *energy impact* when we used alternative Collections instances, while for RQ2, we report that the *changes that we made* were deliberately done (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 [16]. 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 6.1). 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 6.1 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 the results that we achieved for Gson.

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, while we have about 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, we selected a subset of tests for our study (71%).

Out of a total of 258 instances of ArrayList in the codebase and tests (Table 6.1), 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. We made the ‘bad’ change.

Energy Impact The modified version consumed 15% more energy (Table 6.1). Again, the changed library runs ≈ 1.2 times slower than the original version, and thereby consumes more energy during the test run.

6.3 XStream

The XStream library can be used to serialize Java objects in XML and deserialize it back [52]. 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 6.1). The modified version runs ≈ 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 exercised, we augmented the original test suite by generating 256 more test cases for the app. We used JTExpert [44] 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 [26]. 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 6.1). We analyzed the program with WALA, and found that there was scope to prepare both a “good” and a “bad” version of the program.

Our WALA program found that 53 instances of ArrayLists 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 6.1 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 store in lists. On the contrary, K-9 Mail tests were dealing with lists of only a few elements. Therefore, the impact was not very large.

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 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 6.1). The modified version of the library ran ≈ 1.02 times faster than the original, which is probably why the energy consumption was lesser.

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 program 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 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 ≈ 1.6 times faster.

6.7 Discussion

For all our test applications, we were able to get differences in energy consumption by changing the Collections instances (RQ1). The magnitude of change, however, depends on how aggressively the instances are exercised during program execution. With significant usage, we can get large changes in energy consumption when we make bad choices for Collections instances (RQ2).

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 ≈ 4 times slower and consumed ≈ 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 takes. This may explain why it consumes more energy. We found a similar trend in execution times while generating our profiles. But is it a consistent trend that slower applications will consume more energy? More investigation is needed to answer this.

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 anyways. In the future, we want to focus on developing a more sophisticated heuristic to handle other Collections.

Chapter 7

Conclusions

Our results provide a guideline about the scenarios in which the energy consumptions of alternative Collections classes become an issue. For insertion operations, the energy differences are significant, but not that much for other list operations. 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.

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 nondeterminism. Our test-bed was designed to minimize such noise and to control for nondeterministic differences in measurements by repeating measurements 20 times. Section 3.3.2 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. Our results reflect the energy performance of the Android implementation and the Dalvik VM execution of the Collections API. 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.

Our work can be improved in a number of ways. Other API methods, such as `contains()` and `remove()` from the Collections interface, can be studied to better cover the usage scenarios in real applications. Our bytecode analyzer can be enhanced with more sophisticated libraries to trace bytecode traces for Map and Set implementations. The WALA analyzer that we used to discover how Collections instances are used is yet to implement one of the four cases of inter-procedural analysis. Future versions of the work can add this into the library.

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. This study should motivate future work on creating better guidelines for many other alternative programming choices.

Bibliography

[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] M. Fowler. *Refactoring: improving the design of existing code.* Pearson Education India, 2009.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] Java parser. `https://github.com/javaparser/javaparser`.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

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

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

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

[51] 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.

[52] Xstream. `http://xstream.codehaus.org/`.

[53] 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.

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

[55] 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.