2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Recommending Energy-Efficient Java Collections

Wellington Oliveira*, Renato Oliveira*, Fernando Castor*, Benito Fernandes*, Gustavo Pinto†
*Federal University of Pernambuco
{woj, ros3, jbfan, castor}@cin.ufpe.br
†Federal University of Pará
gpinto@ufpa.br

***Abstract*—Over the last years, increasing attention has been given to creating energy-efficient software systems. However, developers still lack the knowledge and the tools to support them in that task. In this work, we explore our vision that energy consumption non-specialists can build software that consumes less energy by alternating, at development time, between third-party, readily available, diversely-designed pieces of software, without increasing the development complexity. To support our vision, we propose an approach for energy-aware development that combines the construction of application-independent energy profiles of Java collections and static analysis to produce an estimate of in which ways and how intensively a system employs these collections. By combining these two pieces of information, it is possible to produce energy-saving recommendations for alternative collection implementations to be used in different parts of the system. We implement this approach in a tool named CT+ that works with both desktop and mobile Java systems, and is capable of analyzing 40 different collection implementations of lists, maps, and sets. We applied CT+ to twelve software systems: two mobile-based, seven desktop-based, and three that can run in both environments. Our evaluation infrastructure involved a high-end server, a notebook, and three mobile devices. When applying the (mostly trivial) recommendations, we achieved up to 17.34% reduction in energy consumption just by replacing collection implementations. Even for a real world, mature, highly-optimized system such as Xalan, CT+ could achieve a 5.81% reduction in energy consumption. Our results indicate that some widely used collections, e.g., `ArrayList`, `HashMap`, and `HashTable`, are not energy-efficient and sometimes should be avoided when energy consumption is a major concern.**

I. Introduction

The extensive adoption of battery-based devices such as mobile phones, smart watches, and laptops greatly increased the importance of energy efficiency as a quality attribute to be acknowledged by application software developers [1]. Moreover, recent work by Manotas *et al.* [2] concluded that, based on responses from hundreds of professionals from industry, developers are interested in building more energy-efficient software, even in scenarios where energy is not an obvious concern, such as desktop application. Chowdhury *et al.* [3] pointed that energy-aware projects tend to be larger than the ones that does not take energy into consideration and that the developers responsible to commit energy changes usually are specialists. One consequence of the widespread interest in energy consumption is that many developers who are not specialists also want their applications to consume less energy. These developers have ample knowledge and experience with their development platforms of choice, but lack both knowledge and tools when it comes to making applications energy-efficient [4]. In this paper, we cater to developers facing this challenge.

Non-trivial software systems have parts where it is possible to use different collections, API calls, concurrency control mechanisms, and libraries by means of simple source code transformations. We call these parts *energy variation hotspots* when these transformations can lead to reduced energy consumption. For example, the Java language has 9 different concrete implementations of hash tables, with different guarantees in terms of scalability, thread-safety, and memory efficiency. Previous work [5] has shown that an insertion operation in one implementation, `ConcurrentHashMap`, can consume less than 1/3 of the same operation in another implementation, `Hashtable`. This means that replacing one implementation by the other can lead to energy savings. Moreover, this result also applies to other languages [6], API usage [7], types of constructs [8], and usage scenarios [9].

Unlike low-level abstractions, such as voltage and frequency scaling, energy variation hotspots are familiar to developers. Moreover, they make it easy to experiment with different options to analyze their impact on energy, since there are readily-available alternative implementations. Furthermore, the cost of replacing one implementation by another tends to be low, since they usually share common specifications. For example, in the aforementioned study [5], replacing uses of `Hashtable` by `ConcurrentHashMap` required changing a single line of code per `Hashtable` object in most cases. Analogously, in a study targeting Haskell's thread-management constructs [8], replacing uses of Haskell's default thread instantiation primitive, `forkIO`, by an alternative, `forkOn`, that binds the created thread to a specific processor required modifications to a single line of code per use of `forkIO`. `forkOn` exhibited lower energy consumption in most of the experiments.

Here we share our vision about solutions for analyzing the energy behavior of alternative collection implementations in a manner that is application-independent and friendly for non-specialists. We propose an approach for energy-aware development that combines the construction of application-independent energy profiles [9] of Java collections and static analysis to produce an estimate of in which ways and how intensively a system employs these collections. By combining these two pieces of information, it is possible to produce energy-saving recommendations for alternative collection implementations to be used in different parts of the system. We have instantiated this approach in a tool named CT+. This tool works in two steps. First, it automatically runs multiple micro-

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00033

benchmarks in an application-independent manner for the Java collections available in a certain execution environment. With data from these micro-benchmarks, it builds energy consumption profiles [9] for the implementations of these collections. An energy profile provides us with a grade that can be used to compare the energy consumption of different implementations of the same abstract operation. After building the energy profiles, the second step consists of performing a static analysis on the system to be optimized, so as to estimate the frequency of use of multiple collection operations. The third step consists of recommending the most efficient implementation for each energy variation hotspot. Finally, CT+ can optionally apply the recommended changes automatically.

We applied CT+ to 12 software systems, seven targeting a desktop environment, two targeting a mobile environment, and three that work in both. With no prior knowledge of the application domains or the system implementations, it was possible to reduce the energy consumption of a software system up to 17.34% just by replacing collection implementations. Even for a real world, mature, highly-optimized system such as XALAN, CT+ could achieve a 5.81% reduction in energy consumption by employing this automated approach. The results of our study highlight the need to re-assess the adoption of some widely popular, poorly-optimized collections from the Java Collections Framework, such as `ArrayList`, `Hashtable`, and `HashMap`. Recommendations to replace uses of these collections by more efficient alternatives were common in our evaluation. In addition, there was not a single case where either of the latter two was recommended. The data related to this work can be found at https://energycollections.github.io/.

This study is an improved version of a previously published short paper [10]. Section III provides more details on the improvements made for this version.

II. JAVA COLLECTIONS

Collections are widely used in applications, on mobile and desktop environments. Java's collections are usually subdivided in three different APIs: `Lists`, `Maps`, and `Sets`. These categories are different in several points but the main factors that distinguish them are: **Lists** are ordered and indexed (with possible duplicates), **Sets** are unordered and do not admit duplicates, and **Maps** are based on key-value pairs and hashing (keys are unique but values can be duplicated). In Java, each collection has multiple implementations.

Collections can be implemented in a number of different ways and can have a non-negligible impact on energy consumption. The usual way to use collections in Java is through the Java Collections Framework (JCF). Yet, previous work [5], [9], [11] has shown that alternative implementations can have a positive impact on the energy consumption of applications. Based on that, for this research we are looking at collections from three different sources: Java Collections Framework [12], Apache Commons Collections [13], Eclipse Collections [14]. To get a glimpse at the usage of these alternative implementations in Java projects, in January 2019 we executed a query on GitHub based on the package names of Eclipse Collections (`org.apache.commons.collections`) and Apache Common Collections (`org.eclipse.collections`). The results showed that these collections are in widespread use, with 1,022,778 occurrences for Apache Common Collections and 466,394 for Eclipse Collections.

TABLE I
JCF COLLECTIONS ACROSS GITHUB JAVA PROJECTS.

Thread Safety	Package name (`java.util`)	Occurrences
Unsafe	`.ArrayList`	35,278,092
	`.HashMap`	16,602,391
	`.HashSet`	6,470,505
	`.LinkedList`	3,763,660
	`.LinkedHashMap`	1,470,500
	`.TreeMap`	1,122,886
	`.TreeSet`	950,890
	`.LinkedHashSet`	689,397
	`.WeakHashMap`	271,852
	Sum of thread unsafe collections	**66,620,173**
Safe	`.Vector`	4,731,762
	`.Hashtable`	1,994,173
	`.concurrent.ConcurrentHashMap`	1,119,704
	`.concurrent.CopyOnWriteArrayList`	237,541
	`.concurrent.CopyOnWriteArraySet`	70,680
	`.concurrent.ConcurrentSkipListMap`	39,012
	`.concurrent.ConcurrentSkipListSet`	26,826
	Sum of thread safe collections	**8,219,698**

Collection implementations that can be safely used by several concurrent threads are considered to be "thread-safe". This safety usually comes with extra complexity or inferior performance, which might favor the use of "thread-unsafe" collections. In this work, we consider that it is never acceptable to replace a use of a thread-safe collection implementation by a thread-unsafe one. Conversely, although it is possible to replace a use of a thread-unsafe collection implementation by a thread-safe one, this is not efficient in practice [5].

JCF collections, both thread-safe and thread-unsafe, are in widepspread use. We conducted another simple query on the adoption of our selected collection implementations on Github Java projects. The results suggest that thread-unsafe collections are used more often than thread-safe collections by a fair margin (66,620,173 and 8,219,698 occurrences, respectively). The most widely used unsafe collection, `java.util.ArrayList`, is used more than four times the sum of all thread-safe collection uses. Table I summarizes the results of our queries.

Creating thread-safe collections based on thread-unsafe collections using the JCF is straightforward: one just needs to use specific `static` methods from the `Collections` class to create `synchronized` Lists, Maps, and Sets. In this work, we labeled those wrapped, thread-safe collections as follows: "Synchronized" + *original collection name*, e.g., `SynchronizedArrayList`.

III. RELATED WORK

We group the related work on the energy efficiency of collection implementations in terms of empirical studies and recommendation tools.

Empirical studies. Hasan *et al.* [9] compared the energy

consumption of collections in Java. They built an energy consumption profile for each collection they analyzed, aiming to answer which implementation of each collection (`Lists`, `Sets`, and `Maps`) consumed less energy. They used that information to manually improve the efficiency of a set of selected applications. Pinto *et al.* [5] studied the thread-safe Java Collections on two different desktop machines. The authors found that the cost of each operation varies widely among different implementations of the same collection. For instance, the authors found energy improvements of 66%, when changing to a more energy efficient implementation of a map. Saborido *et al.* [15] compared two Android-specific collection implementations of `Maps`: `SparseArray` and `ArrayMap`. These implementations were developed to be more efficient than `HashMap`. In summary, `ArrayMap` was considered worse than `HashMap` when optimizing energy consumption and `SparseArray` was considered better when the keys are primitives types.

Here we investigate the impact of software constructs in energy consumption while also offering a recommendation tool that can be used by developers to save energy. Therefore, this work builds upon knowledge produced by previous studies to make recommendations in an automated manner.

Recommendation tools. Manotas *et al.* [16] developed a general purpose framework called SEEDS to guide developers on the laborious work of creating energy-aware software systems. They instantiate the concepts of that framework with the objective of analyzing the consumption of different collection implementations from the JCF. While our proposal uses the concept of energy profile and static analysis to analyze the applications and suggest an implementation of a collection, SEEDS leverages dynamic analysis, executing each different collection for every application and comparing their energy consumption. Furthermore, it did not consider the impact of multithreading and only targeted a desktop environment.

Pereira *et al.* [17] implemented an energy-aware tool called jStanley, aiming to recommend the best collection implementation among several over the JCF. jStanley was implemented as an Eclipse plugin and works using experimental results from their previous work [18]. It does not account for the impact of loops, does not work on a desktop environment, and does not differentiate thread-safe and thread-unsafe collections.

This work is an improved version of our previous work with energy consumption of JFC [19]. We improved on this work by several different means such as: (i) Support to mobile applications; (ii) More operations (19 vs 9) and more collection implementations (40 vs 11) were used to help better analyze the energy consumption; (iii) Support to thread-unsafe collections; (iv) More robust intraprocedural static analysis; (v) Two alternative sources of implementations to the JCF; and, (vi) Twelve software systems (vs two) were analyzed.

IV. Overview of the Proposed Approach

The proposed approach is organized in three phases: (1) creation of the energy profiles, (2) collection usage analysis, and (3) recommendation of source code modifications that have the potential to reduce energy consumption. Figure 1 provides an overview of our approach. In this section, we focus on its high level explanation. Then in Section V we present our instantiation of this approach in the CT+ tool. The three phases are detailed next.

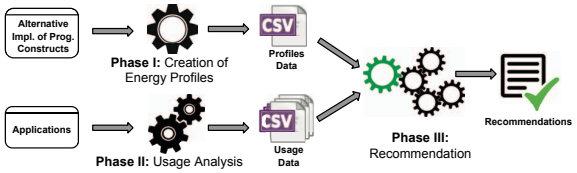

Fig. 1. An overview of our approach. Phase I is application-independent whereas phases II and III also use information about the system under analysis.

Phase I: Creation of Energy Profiles. Here we select a group of programming constructs to analyze and build their energy profiles. This selection determines the energy variation hotspots of the applications that will be analyzed in phase II. Good choices are constructs that are used intensively and that have alternative implementations. As mentioned before, collections, concurrency control mechanisms, and APIs are examples of potential candidates.

Having selected the candidate constructs and their alternative versions, it is necessary to build their energy profiles [9]. An energy profile for a construct is a number that can be used to compare it to similar constructs under the same circumstances. This idea can be explored in diverse situations. For example, previous work [20], [21], [22] computed energy profiles for the colors that can appear in an OLED screen and employed this information to suggest color schemes for smartphone applications that spend less energy. This improvement could be achieved without the need to precisely measure the amount of energy consumed by each color individually.

Energy profiles can be produced by executing several micro-benchmarks to collect information about the energy behavior of these programming constructs in an application-independent way. This step needs only to be performed once for a given construct, per execution platform. The results can then be reused across multiple software systems employing these constructs. In Section V we define energy profiles in a more precise manner for the specific context of collection implementations.

The energy profile created on phase I is used as input to make the recommendation in phase III.

Phase II: Collection Usage Analysis. This phase extracts information about how the target software systems use the selected programming constructs, for example, usage context and frequency of use. This information can be extracted either dynamically or statically. In our instantiation of this approach, we relied on a purely static approach. This has the advantage of being platform-independent and not requiring multiple executions of the system under analysis. On the other hand, it is more prone to imprecision, since it is not possible to know how often an operation will be executed until the system is actually executed.

The output of this step is heavily dependent on the kind of

construct, such as: package, method, class, and line of code where the construct is used, the amount of times its instantiated or invoked, thread running the programming construct, if the invocation of the programming construct is placed inside a loop, among many others. The data collected from phase II is used as input to make the recommendation in phase III.

Phase III: Recommendation. This phase combines the energy profiles and the results of the usage analysis, using the data from phases I and II as input. Different formulae can be employed in this phase. A straightforward approach is to linearly combine the energy profiles with the frequency of use of each alternative constructs. Each of these combinations will yield an energy consumption number that can be directly compared to determine the most energy-efficient alternative. This is the approach we employed in our experiments. We make it more concrete in the next section.

V. Instantiation for Java Collections

Our approach focuses on Java collections and is instantiated in a tool named CT+. CT+ analyzes Java programs in desktop and mobile environments, and recommends collections that could potentially reduce energy consumption.

Phase I: Creation of Energy Profiles. In this phase, we build the energy profiles of the collections. These profiles are based on implementations and operations of three kinds of collections: lists, maps, and sets. We use interchangeable collection implementations for each kind of collection and their operations to create energy profiles that allow these implementations to be compared from an energy consumption standpoint. In this section, we define a collection implementation C abstractly as a tuple $(N, T, S, o_1, o_2, ..., o_n)$, where N is the name of the collection, T is the type of the collection, with $T \in \{List, Set, Map\}$, $S \in \{ThreadSafe, NonThreadSafe\}$, and o_i, $1 \leq i \leq n$ are the n operations of the collection implementation. In this context, an energy profile for a collection implementation is an n-tuple, where each of its elements is a number that can be used to compare the energy consumption of the same operations for different collection implementations under the same execution environment. An energy profile for a collection implementation C is produced by a profiler, a simple function that, in a given execution environment and under a set of workloads, produces an energy profile:

$$profiler(C, env, w_1, w_2, ..., w_n) = (e_1, e_2, ...e_n)$$

where env abstractly represents the execution environment in which the profiler is running (machine, operating system, JVM version), $w_i, 1 \leq i \leq n$, represents the workload applied to each operation o_i in C, and e_i is the energy consumption value for operation o_i. Energy profiles for two collections C_a and C_b can be compared as long as $C_a.T = C_b.T$ and $C_a.S = C_b.S$. In other words, we cannot, for example, compare profiles for a list and a set, nor compare the profile for a thread-safe collection to the profile of a non-thread-safe collection. We assume that collections whose T and S elements are equal have the same operations and, from an implementation standpoint, are functionally equivalent. This is true in practice for the vast majority of the collection implementations in the JCF, with very few exceptions, e.g., `WeakHashMap`. Table II presents all implementations we analyzed in this work.

We build energy profiles by running micro-benchmarks applied to the operations of each collection implementation. The executions were made in a specific cycle of operations. We first perform insertions, then we iterate over the whole collection, and finally we remove all elements previously stored in the collection. In cases where more than one type of insertion or removal is necessary, e.g., lists, where it is possible to insert at the start or end, we pair the classified insertions and removals before the initial sequence (e.g., "insert (start)" and "removal (start)"). The energy consumption was collected throughout each operation. We used this approach to make sure that removals and iterations are measured without the overhead imposed by an insertion operation.

As mentioned above, on lists it is possible insert or remove an element at a specific position, differently from a map or a set. To reflect this behaviour, we distinguish operations to insert or remove list elements at the start, middle, or end of the list. We also consider the "default" operation for insertion or removal. For example, for insertions, it is the `add` method. We iterate over lists using three different approaches: a seeded randomly generated number as index, an explicitly created iterator, and a `for` loop. Table III contains a summary of the operations we analyze to build the energy profiles. Although there are other possible operations (e.g., `List.removeAll()`), they were not used on this study.

To execute the micro-benchmarks and build the energy profiles we developed two different energy profilers, one for the desktop environment and one for the mobile environment. We had to create two different applications because these environments use different methods for gathering energy data and are implemented in different platforms. Nevertheless, we employed the same methodology to collect the energy consumed by each operation on a specific collection implementation. In addition, both profilers were built according to the recommendations of Georges et al. [23] for Java performance evaluation. The **Desktop Profiler** is a simple Java program that uses the jRAPL [24] library to measure energy consumption on desktop applications. It works on Intel architectures (Sandy Bridge and later). The **Mobile Energy Profiler** comprises two subsystems: an Android app, responsible for executing the micro-benchmarks for each operation of each collection implementation, and a dashboard application, responsible for collecting and storing the produced data. We used the Android Power Profiler to measure energy consumption on the mobile applications. That limits our profiler implementation to only work on Android devices with version 5 or later.

The energy profilers containing the energy cost of each operation will be used on Phase III to make recommendations.

Phase II: Collection Usage Analysis. CT+ employs an inter-procedural dataflow static analysis to gather information about the frequency of use and the context in which the collection operations are invoked. For a given program starting point, e.g., the `main` method, it analyzes all the paths in the

TABLE II
THE SELECTED IMPLEMENTATIONS TO BE USED IN THE EXPERIMENTS. THREE DIFFERENT SOURCES WERE USED: JAVA COLLECTIONS FRAMEWORK, ECLIPSE COLLECTIONS AND APACHE COMMONS COLLECTIONS

Collection	Thread Safety	Implementations
List	Safe	Vector, CopyOnWriteArrayList, SynchronizedArrayList, SynchronizedList, and SyncronizedFastList.
	Unsafe	ArrayList, LinkedList, FastList, CursorableLinkedList, NodeCachingLinkedList, and TreeList.
Map	Safe	Hashtable, ConcurrentHashMap, ConcurrentSkipListMap, SynchronizedHashMap, SynchronizedLinkedHashMap, SynchronizedTreeMap, SynchronizedWeakHashMap, ConcurrentHashMapEC, SynchronizedUnifiedMap and StaticBucketMap.
	Unsafe	HashMap, LinkedHashMap, TreeMap, WeakHashMap, UnifiedMap, HashedMap,
Set	Safe	ConcurrentSkipListSet, CopyWriteArraySet, SetConcurrentHashMap, SynchronizedHashSet, SynchronizedLinkedHashSet, SynchronizedTreeSet, SynchornizedTreeSortedSet and SyncronizedUnifiedSet.
	Unsafe	HashSet, LinkedHashSet, TreeSet, TreeSortedSet, and UnifiedSet.

program method call graph that can be reached from there. This analysis aims to identify calls to collection operations that appear within loops, including loops from different methods. One operation can be counted more than once, depending on its context. For example, the method "`bar()`" can be called from the method "`foo()`" in different parts of the code. One invocation of "`bar()`" might be inside a loop while another one may not be involved in loops.

Taking loops into account is important because operations inside them are usually executed several times and thus consume more energy than ones not invoked within loops. In our implementation, we use the nesting level of the loops as a heuristic to give weights to the operations that are performed within them. Even though there are some approaches to determine loop bounds (*e.g.,* [25]), these works (1) do not cover all loop usage scenarios for languages such as Java, where arrays can be allocated dynamically, and (2) typically require program execution. We then opted to use a more conservative approach and only take into account the nesting level of the loops.

Phase III: Recommendation. For each collection implementation object in the code of a program under analysis, our tool makes its recommendation using (i) the energy profile information for that collection implementation (ii) the number of occurrences of the collection operations in the source code related to that specific object, (iii) and whether those occurrences appear within loops or not. More specifically, for each object c which is an instance of a collection implementation C and each path in the program call graph where an operation on c is invoked, CT+ calculates its energy factor EF according to the following simplified formula, considering every collection implementation C' such that $C.T = C'.T$ and $C.S = C'.S$:

$$EF(C', c) = \begin{array}{l} \sum_{i=1}^{n} e_i * NL(c.o_i) + \\ \sum_{j=1}^{n} \sum_{d=1}^{m} e_j * ((L_d(c.o_j) + 1)^{d+1} - 1) \end{array}$$

In this formula, n is the number of operations in C, e_i is the i^{th} element of the energy profile of C', notation $c.o_i$ indicates operation o_i from collection implementation C' invoked at object c, NL is a function that yields the number of non-loop occurrences of o_i targeting c for every path in the program call graph, L_d yields the number of occurrences of o_j applied to c appearing within a loop of nesting level d for every path in the program call graph, and m is the maximum loop depth of the program. The nesting level of a loop affects the energy factor by increasing the exponent which dictates the weight of operations appearing within loops. The "+1" in the exponent and in the innermost summation guarantee that operations appearing within a loop always have a greater weight that those that do not. The "-1" within the innermost summation guarantees that operations not appearing within loops ($L_d(c.o_j) = 0$) get cancelled out.

CT+ makes its recommendation based on the energy factors of the collection implementations. It gives as output an ordered list of collection implementations with better energy footprint than the original collection.

TABLE III
OPERATIONS USED ON EACH COLLECTION.

Collection	Operation	Types
List	insertions	default, start, middle, and end
	iterations	random, iterator, and loop
	removals	default, start, middle, end, and object
Map	insertions	default
	iteration	iterator and loop
	removal	default
Set	insertions	default
	iteration	loop
	removal	default

Implementation. The implementation of CT+ is based on WALA [26], a static analysis library developed by IBM. Since a collection may be thread-safe or not, we employ WALA's built-in type inference and points-to analysis to discover the concrete types of objects. This is also useful to support recommendations that account for collection objects being passed as method arguments.

VI. EVALUATION

We applied our evaluation to a number of software systems, running on multiple execution environments, comparing the energy consumption of the original version with a modified one, according to the recommendations produced by our tool.

A. Research Questions

Among Java's diverse collection implementations, developers may opt to use the most popular ones, even though popularity is not necessarily a proxy to energy efficiency. To investigate address this issue, we experimented with several collections available in the JCF, as well as several alternative implementations. Furthermore, our approach is based on the assumption that the energy profiles of the analyzed collections can be different depending on the underlying execution platform. This concern is particularly important due to the great diversity of devices and operating system versions available for use. Based on these considerations, with this study we aim to answer the following research questions (RQs):

RQ1: To what extent can we improve the energy efficiency of an application by statically recommending Java Collections?

RQ2: Are the recommendations device-independent?

B. Methodology

Our evaluation comprises two different execution environments, **desktop** and **mobile**. These environments differ in terms of the available processing power and memory, use of batteries, and measurement procedure. We employ jRAPL to perform energy measurements in the desktop environment and the Android Energy Profiler in the mobile environment (Section V). Table IV presents a summary of the devices used in this study. On the desktop environment, we executed CT+ across two different machines, a notebook (**note**) with an Intel Core i7-7500U with four 2.7GHz cores, and 16GB of RAM and a high-end server (**server**) with a a two-node Intel Xeon E5-2660 v2 processor with 20 2.20GHz cores (10 per node) and 256GB of RAM. As for mobile devices, we executed our tool on three smartphones: a Samsung Galaxy J7 (**J7**), a Samsung Galaxy S8 (**S8**), and a Motorola G2 (**G2**).

When creating the profiles, we choose to use the same methodology as previous work [6], [9], although we leverage a larger number of collection implementations (Table II). We executed the micro-benchmarks, each one representing an operation-collection pair, and calculated their energy consumption.This procedure is repeated 30 times for each micro-benchmark, for each machine. To reduce the interference of the JIT, we performed a warmup execution. In the warmup, we executed up to 10% of our workload. We only started collecting the samples after the end of the warmup period. By doing this, we minimized JIT noise on the measurements [27].

We analyzed seven desktop-based software systems: BARBECUE, BATTLECRY, JODATIME, TOMCAT, TWFBPLAYER, XALAN, and XISEMELE; two mobile-based software systems: FASTSEARCH and PASSWORDGEN; and three on both environments: APACHE COMMONS MATH 3.4 (COMMONS MATH for short), GOOGLE GSON, and XSTREAM: These systems were the employed in related work on energy profilling [5], [9], [17], [28], and their workloads are available for replication purposes. For **server**, we only ran TOMCAT and XALAN since these are applications one would expect to execute in a high-end server machine. For TOMCAT, we could not recommend any implementation from the Eclipse Collections library. This happened because the DACAPO suite [29] endorses the use of Java Development Kit (JDK) to ensure the correct operation of their benchmarks. Unfortunately, the current version of Eclipse Collections is incompatible with JDK 6.

For TOMCAT and XALAN on the desktop development machines, we used the same workloads (provided by DACAPO) on both systems, varying only the number of threads for each machine: 40 on **server** and four on **note**. For four systems (BARBECUE, JODATIME, TWFBPLAYER, XISEMELE) we used unitary tests, following the same methodology as previous work [17]. On BATTLECRY, we executed a class inside the benchmark designed to test it. On GOOGLE GSON and XSTREAM we tried to exercise each Java primitive using methods inside those systems. With APACHE COMMONS MATH 3.4 we executed multiple statistical functions from its API. As both PASSWORDGEN and FASTSEARCH are very one dimensional software system, their workload consisted of executing their main function (e.g., generating passwords).

Different devices require different workloads to run for enough time for the energy measurement to have expressive values. This adjustment was specially important when running the mobile profiler. Whereas jRAPL [24] is capable of code-level, fine-grained measurement, the Android battery dump collects battery data at the process level. In order to mitigate potential imprecisions, we adjusted the mobile micro-benchmark executions to run for at least 20 seconds.

For the experiments, we collected the results of 30 executions of each software system. When experimenting with thread-safe collections, we used four threads for each operation; with non thread-safe collections, only one thread was used. Since most of our samples are not normally distributed, based on Shapiro-Wilk's normality test [30], we used the Wilcoxon-Mann-Whitney test [31] to test whether the difference in energy consumption between the original and modified versions of each software system is statistically significant. We did not remove any outliers. We also employed Cliff's Delta [32] as a measure of effect size. Wilcoxon-Mann-Whitney test and Cliff's Delta are non-parametric.

C. Study results

We present the results in terms of the desktop and the mobile environments. For each one, we present first the energy consumption results and then proceed to discuss the recommendations that were made for each software system. We will only present the energy results, because as mentioned in Section VI-B most executions had a designed workload based on the time necessary to execute it. The specific amount of time each system took to be executed can be found at https://energycollections.github.io/.

Desktop environment. Table V summarizes the energy consumption for the desktop environment. The most important column of the table is **Improv**, which shows how much more energy the original version consumed, when compared to the modified one. A positive percentage in this column indicates that the modified version consumes less energy than the original one. The versions of all the software systems modified according to the recommendations of CT+ consumed less energy

TABLE IV
THE MACHINES USED IN THE EXPERIMENTS AND THEIR CHARACTERISTICS

Machine	Alias	RAM	Chipset	CPU	Battery
Notebook	**note**	16GB	i7-7500U	Quad-core 2.70GHz	N/A
Server	**server**	256GB	Intel Xeon E5-2660 v2	40-core 2.2 GHz	N/A
Samsung J7	**J7**	1.5GB	Exynos 7580	Octa-core 1.5 GHz Cortex-A53	3000 mAh
Samsung S8	**S8**	4GB	Exynos 8895 Octa	4x2.3 GHz Mongoose M2 & 4x1.7 GHz Cortex-A53	3000 mAh
Motorola G2	**G2**	1GB	Qualcomm MSM8226 Snapdragon 400	Quad-core 1.2 GHz Cortex-A7	2070 mAh

TABLE V
RESULTS FOR THE DESKTOP ENVIRONMENT. ENERGY RESULTS ARE RED FOR THE ORIGINAL VERSIONS AND GREEN FOR THE MODIFIED VERSIONS.

Device	System	Improv	p-value	Mean(J)	Stdev	Effect Size
note	Barbecue	4.58%	7.0^{-4}	56.17 53.71	2.70 2.53	0.50
	Battlecry	2.86%	1.5^{-3}	67.95 66.06	2.67 3.18	0.48
	Gson	0.72%	8.0^{-5}	29.93 29.72	0.22 0.16	0.57
	Commons Math	1.05%	6.3^{-12}	48.93 48.43	0.29 0.15	0.90
	JodaTime	7.13%	$< 2.2^{-16}$	123.02 114.83	2.42 3.50	0.94
	Tomcat	4.12%	$< 2.2^{-16}$	32.77 31.47	1.02 0.41	0.86
	Xalan	5.01%	$< 2.2^{-16}$	107.04 101.93	0.19 0.15	1
	Xstream	2.58%	3.122^{-13}	59.97 58.45	0.52 0.49	0.94
server	Tomcat	4.3%	$< 2.2^{-16}$	89.21 85.54	2.99 2.37	0.66
	Xalan	5.81%	$< 2.2^{-16}$	242.29 228.98	4.4 7.02	0.86

than the original versions. For two of them, (TWFBPLAYER and XISEMELE), the difference between original and modified versions was not statistically significant. Notwithstanding, for the remaining systems in both the **note** and **server** machines, the difference is statistically significant and effect size is large. According to Romano *et al.* [33], effect size as measured by Cliff's Delta can be considered large when it is ≥ 0.474. In particular, for XALAN in the **note** machine, the effect size was 1, which means that every execution of the modified version exhibited lower energy consumption than every execution of the original version. Among the software systems that only ran in the **note** machine, JODATIME exhibited the greatest improvement, with the original version consuming 7.13% more than the modified one. Hereafter, due to space constraints, this section focuses on the results for which $p-value < 0.05$, thus indicating a statistically significant difference, either positive or negative, between the original version and the modified one. We analyze some cases where the difference was not statistically significant in Section VII.

The two software systems that were executed in the **note** and **server** machines, XALAN and TOMCAT, exhibited positive results in both scenarios. For XALAN, the original version consumed 5.01% and 5.81% more energy than the modified version in the **note** and **server** machines, respectively. For Tomcat, the differences were of 4.12% and 4.3%, respectively. We found that systems running on **server** consumed more than twice the energy they consumed on **note**, for the same workload. This can be justified in terms of their differences in processing power. Notwithstanding, the results were consistent across the two machines.

Table VI summarizes our results for each application on **note** and on **server**. In both machines XALAN had a significant number of instances of `Hashtable` changed to `ConcurrentHashMapEC` (48 and 49 times on **note** and **server**, respectively). In fact, for both **server** and **note**, we can observe a trend of recommendations to replace well-known collections from the JCF (`Vector`, `ArrayList`, `HashMap`) by alternatives from Eclipse Collections and Apache Commons Collections. For the specific case of XALAN, among the 119 recommendations across the two desktop machines, just three were for JCF collection implementations.

TOMCAT recommendations differed across the two machines. On **note**, the tool made 13 energy saving recommendations, seven for collections from the JCF, and six for collections from the Apache Commons Collections. On **server**, there were 60 recommendations, 40 for Apache Commons Collections, and 20 for JCF collections. In particular, there were 68 recommendations to replace `Hashtable`, `HashSet`, or `HashMap` by more energy-efficient alternatives and no recommendation to use any of those. As pointed out in Table I, these are widely-used collections. We reiterate that Eclipse Collections could not be recommended for Tomcat (Section VI-B).

Among the six remaining software systems, there were 285 recommendations. From those, 282 suggested the use of collection implementations not from the JCF, 88 from Apache Commons and 194 from Eclipse Collections. Once again it is possible to observe a trend of replacing well-known collections such as `Hashtable`, `HashMap`, and `ArrayList` by more energy-efficient but less-known alternatives.

Mobile environment. Table VII summarizes the results for the mobile environment. The effectiveness of the recommendations varied strongly among the analyzed devices. The modified versions of PASSWORDGEN on the **S8** and **J7** devices exhibited significant improvements: the original versions consumed 4.7% and 17.34% more energy than the modified ones, with a large effect size. However, **G2** had no recommendations (more on this on Section VII) and thus the results were the same. GSON exhibited a significant improvement of 5.03% on the **J7**, with a medium effect size. Nonetheless, the recommendations of CT+ yielded a statistically significant but small 0.95% improvement on **S8**. COMMONS MATH had more inconsistent results. Although the original version consumed 11.31% more energy than the modified version on **S8**, the

TABLE VI
RECOMMENDED COLLECTIONS FOR **NOTE** AND **SERVER**

System	Original	Recommended	# of times
Development machine: note			
Barbecue	HashMap	HashedMap	13
	ArrayList	FastList	8
Battlecry	LinkedList	ArrayList	2
	LinkedList	FastList	2
Commons Math	ArrayList	FastList	112
	HashSet	UnifiedSet	6
	HashMap	HashedMap	9
	HashMap	UnifiedMap	3
	ArrayList	TreeList	3
Google Gson	ArrayList	FastList	12
	HashMap	HashedMap	3
	ConcurrentHashMap	ConcurrentHashMapEC	1
JodaTime	ArrayList	FastList	8
	HashMap	HashedMap	7
	ConcurrentHashMap	ConcurrentHashMapEC	1
Tomcat	Hashtable	ConcurrentHashMap	6
	HashMap	HashedMap	4
	Hashtable	StaticBucketMap	2
	Vector	Synchronized LinkedList	1
Xalan	Hashtable	ConcurrentHashMapEC	48
	ArrayList	FastList	10
	Vector	Synchronized FastList	3
	ArrayList	NodeCachingLinkedList	1
	HashMap	HashedMap	1
Xstream	HashMap	HashedMap	52
	ArrayList	FastList	21
	HashSet	UnifiedSet	12
	HashMap	UnifiedMap	7
	LinkedList	TreeList	1
	ArrayList	LinkedList	1
	HashSet	TreeSortedSet	1
Development machine: server			
Tomcat	HashMap	HashedMap	39
	Hashtable	ConcurrentHashMap	16
	LinkedList	TreeList	2
	LinkedList	ArrayList	1
	HashSet	LinkedHashSet	1
	Vector	Synchronized ArrayList	1
Xalan	Hashtable	ConcurrentHashMap(EC)	49
	Vector	Synchronized ArrayList	3
	ArrayList	TreeList	2
	HashMap	HashedMap	1
	HashMap	UnifiedMap	1

original versions consumed 1.2% and 0.33% less energy than the modified ones on **G2** and **J7**. Albeit small, these results are statistically significant and the effect size for both cases was negative (medium and large, respectively). This intuitively means that it was more common for executions of the modified versions to exhibit greater energy consumption. Finally, FASTSEARCH was arguably the most consistent of the software systems on the mobile environment, in the sense that there was no practical difference between original and modified versions. For **J7** and **G2** the results for the modified and original versions did not differ in a statitically significantly way. On the **S8**, albeit statistically significant, the difference was small with the original version consuming just 0.09% more than the modified version. These results suggest that (i) the energy consumption of different collection implementations varies considerably across mobile devices, and (ii) although the results were not as strong as in the desktop environment, for most cases the recommendations of CT+ either yielded an improvement or did not have a strong impact on the energy consumption of the software systems.

TABLE VII
RESULTS FOR THE MOBILE ENVIRONMENT. ENERGY RESULTS ARE RED FOR THE ORIGINAL VERSIONS AND GREEN FOR THE MODIFIED VERSIONS.

Device	System	Improv	p-value	Mean(J)	Stdev	Effect Size
S8	Commons Math	11.31%	1.25^{-8}	92.06 82.70	2.59 9.61	0.86
	FastSearch	0.09%	1.67^{-3}	35.06 35.03	3.32 1.78	-0.47
	Google Gson	0.95%	6.42^{-4}	16.45 16.29	0.22 0.20	0.40
	PasswordGen	4.70%	2.38^{-9}	16.86 16.11	0.41 0.65	0.90
J7	Commons Math	-0.33%	2^{-4}	23.82 23.90	2.33 2.62	-0.56
	Google Gson	5.03%	3.2^{-3}	13.78 13.12	1.59 2.67	0.44
	PasswordGen	17.34%	6.44^{-9}	12.83 10.94	0.90 0.76	0.87
G2	Commons Math	-1.21%	0.0091	17.22 17.42	0.51 0.14	-0.41

Table VIII presents the collections recommended for **S8**, **J7**, and **G2**. COMMONS MATH running on the **S8** has more recommendations for JCF collection implementations than all the software systems we evaluated on the **note** machine combined. On the one hand, the only collection recommended by CT+ that is not from the JCF for this software System is `TreeList` from the Apache Commons Collections. On the other hand, it follows the pattern of recommending alternatives to widely popular collections, e.g., it recommends the use of `TreeList` instead of `ArrayList` and `LinkedHashMap` in place of `HashMap`. For the remaining systems, CT+ made few recommendations, 11 for GSON, 2 for PASSWORDGEN, and 5 for FASTSEARCH. Overall, the recommendations only produced a large effect size for COMMONS MATH and PASSWORDGEN. Furthermore, these were the only systems that could achieve energy savings greater than 1% in the **S8**.

Among the 22 recommendations of COMMONS MATH on **J7**, 14 were for Eclipse Collections and eight were for Apache Commons Collections. In all these cases, CT+ recommended that developers replace `ArrayList` by an alternative implementation. For this specific context, the recommendations did not yield energy savings. CT+ also recommended replacing `ArrayList` by alternatives in the case of GSON and PASSWORDGEN. These substitutions yielded considerable energy savings. The **G2** differed from the others in this study in the sense that only one of the software systems exhibited significant differences between the original and modified versions. Notwithstanding, the trend of CT+ recommending less popular collections as replacements for widely-used ones such as `ArrayList` and `HashMap` can still be observed.

VII. DISCUSSION

This section discusses in more depth some of the results presented in Section VI-C.

TABLE VIII
RECOMMENDED COLLECTIONS FOR **S8**, **J7**, AND **G2**

System	Original	Recommended	# of times
Device: S8			
Commons Math	ArrayList	TreeList	8
	HashMap	LinkedHashMap	7
	HashSet	LinkedHashSet	6
	TreeSet	LinkedHashSet	2
	TreeMap	LinkedHashMap	2
	ArrayList	LinkedList	1
Google Gson	ArrayList	FastList	6
	HashMap	LinkedHashMap	3
	ArrayList	TreeList	1
	ConcurrentHashMap	Synch LinkedHashMap	1
PasswordGen	ArrayList	FastList	2
FastSearch	ArrayList	FastList	4
	HashMap	HashedMap	1
Device: J7			
Commons Math	ArrayList	FastList	14
	ArrayList	NodeCachingLinkedList	5
	ArrayList	TreeList	3
Google Gson	ArrayList	FastList	7
	ArrayList	NodeCachingLinkedList	2
PasswordGen	ArrayList	FastList	5
Device: G2			
Commons Math	HashMap	LinkedHashMap	12
	ArrayList	FastList	8
	ArrayList	TreeList	5
	CopyOnWriteArrayList	Vector	1
	ArrayList	LinkedList	1

JCF recommendations. The majority of the CT+ recommendations were for collection implementations not in the JCF. Considering only the statistically significant occurrences, out of 477 recommendations made in the desktop environment, only 31 suggested the use of JCF collections (6.5% of the recommendations). The contrast is less stark in the mobile environment, where CT+ recommended JCF collection implementations in one third of the cases (36 out of 107 recommendations). If we aggregate over all of these recommendations, the JCF was recommended in 11.47% of the cases.

Popular collections and energy efficiency. Our results indicate that there seem to be more energy-efficient alternatives to some widely popular collection implementations. For the desktop environment, CT+ performed 121 recommendations to replace uses of `Hashtable`, 140 for `HashMap`, 20 for `HashSet`, 8 for `Vector`, and 178 for `ArrayList`. Overall, those recommendations amount to 97.9% of all the statistically significant recommendations CT+ made in that environment. This percentage is consistent with the popularity of those JCF collections (Table I); since they are used often, there will be many recommendations to replace them by alternatives. On the other hand, CT+ did not recommend the use of any of these four collection implementations: `Hashtable`, `HashMap`, `HashSet`, and `Vector`. `ArrayList` was recommended three times, all of them as a replacement for `LinkedList`, a collection that is not efficient for random accesses. These results, combined with the significant improvements in energy efficiency that could be achieved by following CT+'s recommendations in the desktop environment, suggest that these collections might not be good choices in scenarios where energy efficiency has a high priority.

As pointed out previously, in the mobile environment CT+ recommended the use of JCF collections more often. Nevertheless, a similar trend can be observed. CT+ suggested alternatives to `HashMap` 23 times, to `HashSet` 6 times, and to `ArrayList` 72 times. That amounts to 94.39% of all its recommendations. At the same time, not once did it recommend the use of these collections. Furthermore, `Vector` was recommended once, as a replacement for `CopyOnWriteArrayList`, a thread-safe collection that is efficient for reads but extremely inefficient for writes [5].

Given the importance of the aforementioned collections, we conducted a more in-depth investigation into why `ArrayList` was replaced an expressive number of times and only rarely recommended. We focus on `ArrayList` because it is arguably the most popular collection implementation in the Java language. Two factors help explain the lack of recommendations in its favor. First, the most common operations in the software systems for list collections are `add(value)` and `iteration(random)`. `ArrayList` does not perform these operations well on most devices. In particular, `FastList` was explicitly designed as an alternative to `ArrayList` that performs those operations more efficiently, since it does not support concurrent modification exceptions. As a consequence, `FastList` can "*provide optimized internal iterators which use direct access against the array of items.*" [14]. This kind of direct access is not allowed by `ArrayList`. Second, there are many cases where `ArrayList` is the most efficient alternative, but it is already being used. That is what occurred, for example, for `FastSearch` and `PasswordGen` in the **G2**. In other words, due to the widespread use of this collection implementation, in most cases where it would be the best option, it is already being employed and thus no benefits can be achieved.

Different devices matter. The recommendations and results varied heavily across devices, even when executing the same application. Although for some specific applications, such as FASTSEARCH, our tool made similar recommendations across devices and those recommendations did not impact energy efficiency, for most software systems different devices resulted in different recommendations. For instance, for XALAN on **note**, CT+ recommended that 10 `ArrayList` instances be changed to `FastList` and one to `NodeCachingLinkedList`. However, for **server**, it made recommendations for only two instances of `ArrayList` and suggested the use of `TreeList`. In both machines, energy consumption decreased.

In addition, the effectiveness of CT+'s recommendations for the same software systems varied across machines. XSTREAM presents an interesting example. The recommendations made by CT+ did not result in a version of the software system that had a statistically significant difference in energy consumption on the mobile devices, even if the modified versions consumed

less energy. On the other hand, on **note**, the energy consumption of the modified version exhibited a statistically significant difference (with a p-value of 3.12^{-13}) when compared to the original version. Also, the effect was large (0.94). This difference may be attributed to the number of implementation changes as well as differences between devices. On **note**, our tool suggested 95 modifications to XSTREAM while the mobile device with most changes, **G2**, only had 41. Those changes also did not target the same implementations; On **note**, we replaced `ArrayList` by `FastList` 21 times and by `LinkedList` one time. On **G2**, `ArrayList` was replaced by `TreeList` just three times. Those devices had different energy profiles and by the number of changes, we noticed that the implementations used on the mobile versions were already optimized for that environment, which was not the case for the desktop environment.

Dominance among collection implementations. Only 20 out of the 40 possible implementations were recommended by CT+. When trying to understand this behavior, we observed that some collection implementations consistently dominate [34] others. Given two collection implementations $C_1 = (N, T, S, o_1, o_2, ..., o_n)$ and $C_2 = (N', T, S, o_1, o_2, ..., o_n)$ with energy profiles $(e_1, e_2, ..., e_n)$ and $(e'_1, e'_2, ..., e'_n)$, respectively, we say that C_1 dominates C_2 if $e_i < e'_i$ for all $1 \leq i \leq n$. Since every dominated collection implementation has a dominating alternative collection implmementation, it will never be recommended by CT+.

Figure 2 depicts dominance relations for the thread-safe `Map` implementations on the **server** machine. Based on this figure, only four thread-safe `Map` implementations can be recommended by CT+ on the **server** machine: `ConcurrentHashMap`, `Synchronized LinkedHashMap`, `ConcurrentHashMapEC`, and `Synchronized UnifiedMap`. These are the collections that are not dominated by any other collection. Furthermore, as the figure shows, `Hashtable` is dominated by `ConcurrentHashMapEC`, even though `Hashtable` itself also dominates `Synchronized TreeMap`. Therefore, in **server**, instances of `Synchronized TreeMap` and `Hashtable` are never recommended, in favor of `ConcurrentHashMapEC`. More specifically, we observed that `Hashtable` was dominated on every device that we experimented with. This result, combined with the well-known scalability limitations of this collection [5], and the plethora of more efficient alternatives suggest that it should rarely be used in practice. Implementations such as `ConcurrentSkipListSet`, `Synchronized TreeMap`, and `Synchronized UnifiedMap`, were dominated in three out of four devices.

Threats to Validity. Although we conducted experiments in a number of different devices, we did not use all possible devices available, which is far from feasible. We selected representative devices with very different hardware characteristics (from a mobile phone with 1.5GB of RAM to a server with 256GB of RAM). Second, our findings cannot be generalized to other software applications that use collections.

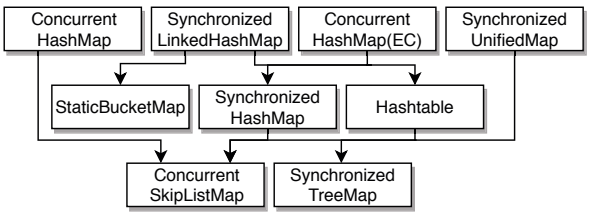

Fig. 2. Order of dominance between the thread-safe Map implementations on **server**. Arrows point from the dominating collection to the dominated one.

We then chose representative software systems from very different domains (e.g., a XML serializer, a webserver, and mobile apps). Still, the chosen software systems are non-trivial, e.g., TOMCAT has more than 640k lines of code and has been used in multiple studies [5], [9], [17]. Even though we observed an overall good energy savings with our tool, for some software systems it was not possible to improve the energy consumption reported in other studies. We hypothesize this happens due to the care we took of preventing recommendations with different thread-safety. We checked that among the recommendations on study [17], there were cases where a thread-safe collection was replaced by a non-thread-safe one. A more comprehensive investigation is necessary. Similarly, our tool does not guarantee thread-safety when thread-safe collections are performing compounded operations [35] (e.g., verifying if an item is stored in a collection before adding it). The software constructs versions (such as libraries and applications) may influence the recommendations made by CT+. The configurations and source code are available at https://energycollections.github.io/. Finally, we did not perform experiments with actual developers, so it is unclear whether developers would face any difficulties while using the tool or whether they would find the recommendations useful.

VIII. CONCLUSION

With this work, we present our vision of a general purpose approach to aid non-specialist developers to create energy-aware software. This vision was instantiated within a tool to recommend energy-efficient collection implmementations. We evaluated our tool in five different devices running twelve different software systems (two mobile, seven desktop, and three on both environments). Although some cases the recommendations provided did not have a direct impact on energy consumption, our tool was able to reduce energy consumption of some applications up to 17.34%. Our results suggest that some of the most popular collections implementations (e.g., `ArrayList`, `HashMap`, and `Hashtable`) are often not the most energy-efficient ones. As *future work* we plan to use our tool in a real world setting to understand whether developers could, indeed, take advantage of it.

Acknowledgments. We would like to thank the anonymous reviewers for helping to improve this paper. This research was partially funded by CNPq/Brazil (304755/2014-1, 406308/2016-0, 465614/2014-0) and FACEPE/Brazil (APQ-0839-1.03/14, 0388-1.03/14, 0592-1.03/15).

REFERENCES

[1] Gustavo Pinto and Fernando Castor. Energy efficiency: a new concern for application software developers. *Communications of the ACM*, 60(12):68–75, 2017.

[2] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *ICSE*, pages 237–248, 2016.

[3] Shaiful Alam Chowdhury and Abram Hindle. Characterizing energy-aware software projects: Are they different? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 508–511, New York, NY, USA, 2016. ACM.

[4] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, May 2016.

[5] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java thread-safe collections. In *ICSME*, 2016.

[6] Wellington Oliveira, Renato Oliveira, and Fernando Castor. A Study on the Energy Consumption of Android App Development Approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[7] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[8] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, March 2016.

[9] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236, New York, NY, USA, 2016.

[10] John Doe-I. Blind. Blind, 2011. Blind.

[11] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 389–400, New York, NY, USA, 2017. ACM.

[12] Oracle Corporation. JFC. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/. Last access: 2019-01-22.

[13] The Apache Foundation. Apache Collections. https://commons.apache.org/proper/commons-collections/. Last access: 2019-01-22.

[14] The Eclipse Foundations. Eclipse Collections. https://www.eclipse.org/collections/. Last access: 2019-01-22.

[15] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Getting the most from map data structures in Android. *Empirical Software Engineering*, March 2018.

[16] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, 2014.

[17] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. jStanley: Placing a green thumb on java collections. In *Proceedings of the 33rd*

[20] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 143–154, New York, NY, USA, 2015. ACM.

ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pages 856–859, New York, NY, USA, 2018. ACM.

[18] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pages 15–21, New York, NY, USA, 2016. ACM.

[19] Benito Fernandes, Gustavo Pinto, and Fernando Castor. Assisting non-specialist developers to build energy-efficient software. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 158–160, May 2017.

[21] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, 27(3):14:1–14:47, 2018.

[22] Mian Wan, Yuchen Jin, Ding Li, Jiaping Gui, Sonal Mahajan, and William GJ Halfond. Detecting display energy hotspots in android apps. *Software Testing, Verification and Reliability*, 27(6):16–35, 2017.

[23] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[24] Kenan Liu, Gustavo Pinto, and David Liu. Data-oriented characterization of application-level energy optimization. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, FASE'15, 2015.

[25] Raphael Ernani Rodrigues, Péricles Alves, Fernando Pereira, and Laure Gonnord. Real-world loops are easy to predict: a case study. In *Workshop on Software Termination (WST'14)*, 2014.

[26] IBM. IBM T. J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page. Last access: 2019-01-22.

[27] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, October 2017.

[28] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, 2014.

[29] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[30] S. S. Shapiro and M. B. Wilf. An Analysis of Variance Test for Normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[31] Daniel S. Wilks. *Statistical methods in the atmospheric sciences*. Elsevier Academic Press, Amsterdam; Boston, 2011.

[32] N. Cliff. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin*, 114(3):494–509, 1993.

[33] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, 2006.

[34] Martin Peterson. *Decisions under ignorance*, page 4063. Cambridge Introductions to Philosophy. Cambridge University Press, 2009.

[35] Yu Lin and Danny Dig. A study and toolkit of CHECK-THEN-ACT idioms of java concurrent collections. *Softw. Test., Verif. Reliab.*, 25(4):397–425, 2015.