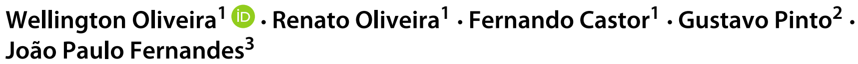
Empirical Software Engineering (2021) 26: 55
https://doi.org/10.1007/s10664-021-09950-y

Improving energy-efficiency by recommending Java collections


Wellington Oliveira[1] · Renato Oliveira[1] · Fernando Castor[1] · Gustavo Pinto[2] · João Paulo Fernandes[3]

Accepted: 4 February 2021 / Published online: 12 April 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC part of Springer Nature 2021

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 non-specialists can build software that consumes less energy by alternating 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. We implement this approach in a tool named CT+ that works with both desktop and mobile Java systems and is capable of analyzing 39 different collection implementations of lists, maps, and sets. We applied CT+ to seventeen software systems: two mobile-based, twelve desktop-based, and three that can run in both environments. Our evaluation infrastructure involved a high-end server, two notebooks, three smartphones, and a tablet. Overall, 2295 recommendations were applied, achieving up to 16.34% reduction in energy consumption, usually changing a single line of code per recommendation. Even for a real-world, mature system such as `Tomcat`, CT+ could achieve a 4.12% 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.

Keywords Energy consumption · Collections · Recommendation systems

1 Introduction

Responsible energy consumption is a problem that permeates most modern human activity. It is not by chance that three of the UNs sustainable development goals can be linked

Communicated by: Ali Ouni, David Lo, Xin Xia, Alexander Serebrenik and Christoph Treude

This article belongs to the Topical Collection: *Recommendation Systems for Software Engineering*

✉ Wellington Oliveira
woj@cin.ufpe.br

Extended author information available on the last page of the article.

with better usage of energy supplies[1]. Energy became a particular problem for the IT industry with the extensive adoption of battery-based devices such as mobile phones, smart watches, and laptops, in conjunction with the already very power-hungry data-centers. If left unchecked, just the data-centers alone could consume up to 20% of global electricity and be responsible for 5.5% of the global greenhouse gas emissions by 2025, with the latter going up as high as 14% by 2040 (Andrae 2017). This clearly conflicts with the ambitious goals of reducing emissions by 40% by 2030, aiming for neutrality by 2050[2], set by the President of the European Commission. In fact, it is anticipated that the energy consumed by IT devices and services in a globally connected world will soon have a bigger impact on global warming than the entire aviation industry (Mingay 2007).

Developers have been more aware of the importance of energy for the environment and end-users, and are interested in building more energy-efficient software (Manotas et al. 2016). Notwithstanding, thinking about algorithms and solutions that are energy efficient is not a typical skill for a developer. In addition, some solutions that developers typically employ or recommend are not backed up by scientific evidence (Pinto et al. 2014a). As pointed out by previous work (Chowdhury and Hindle 2016), energy-aware projects usually are larger and the changes that could impact the application's energy efficiency are most of the times relegated to specialists.

Building an application can be a complex task, requiring developers to deal with interconnected and sometimes conflicting objectives in order to solve non-trivial problems. One way to mitigate this complexity is to leverage the availability of software solutions such as libraries, APIs, frameworks, gists[3], and answers from Q&A sites such as StackOverflow[4]. In this context, designing and implementing software has become a task of selecting appropriate solutions among multiple options (Baldwin and Clark) and combining them to build working systems. We call *energy variation hotspots* the programming constructs, idioms, libraries, components, and tools in a system for which there are multiple, interchangeable, readily-available solutions that have potentially different energy footprints.

Several previous papers have measured and analyzed different types of energy variation hotspots, such as programming languages (Oliveira et al. 2017; Pereira et al. 2017; Georgiou and Spinellis 2020), API usage (Aggarwal et al. 2014; Linares-Vásquez et al. 2014; Rocha et al. 2019), thread management constructs (Pinto et al. 2014b; Lima et al. 2016), data structures (Hasan et al. 2016; Lima et al. 2016; Pereira et al. 2016; Pinto et al. 2016), color schemes (Li et al. 2014b; Linares-Vásquez et al. 2015), and machine learning approaches (Mcintosh et al. 2019), among many others.

Unlike low-level abstractions, such as voltage and frequency scaling, developers are familiarized with energy variation hotspots. 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 (Pinto et al. 2016). If these algorithms adhere to a common specification, a recommendation tool can analyze their usage context so as to recommend a potentially efficient alternative. This can yield

[1] ONU sustainable goals: https://sdgs.un.org/goals

[2] EU guidelines: https://ec.europa.eu/commission/sites/beta-political/files/political-guidelines-next-commission_en.pdf

[3] Gist website: http://gist.github.com

[4] StackOverflow website: https://stackoverflow.com

energy savings at a cheap cost in terms of development effort and does not require specialist knowledge.

In this paper, we share our vision of a solution to help investigate the energy behavior of energy variation hotspots within an application and, when possible, make recommendations that can reduce its energy consumption. The solution we propose to support non-specialists reduce the energy consumption of an application comprises three steps. First, the available alternative solutions are exercised to build execution environment-specific energy consumption profiles (Hasan et al. 2016). These profiles provide a mean to compare the energy footprint of these solutions in an application-independent manner. Second, the application is analyzed to gather information about the selected energy variation hotspots, in particular, to estimate how intensively the system uses them. That step is device-independent. Finally, these two pieces of information are combined to make potentially energy-saving recommendations specific to the application-device pair.

We have instantiated our approach in a tool named CT+. Its goal is to optimize the energy consumption of Java collections on desktop and Android applications. Implementing the first step of our approach, it automatically runs multiple micro-benchmarks for 39 different Java collection implementations in an application-independent manner and builds their energy profiles. `List`, `Set`, and `Map` are the three collections targeted by these collection implementations. The latter stem from the Java Collections Framework (25 implementations), Apache Commons Collections (5 implementations), and Eclipse Collections (9 implementations). With data from these micro-benchmarks, it builds the energy consumption profiles. For the second step, an inter-procedural static analysis is performed on the application source code. This analysis collects, for each instantiation of a collection implementation, information such as frequency and location of use, method and variable names, calling methods, etc. The third and final step consists of recommending the most efficient implementation for each energy variation hotspot, considering both the energy consumption of the multiple operations of each collection implementation and the extent to which each collection is used in the application. The recommendations made by CT+ are applied automatically.

We evaluated CT+ in two studies aiming to answer the following four research questions:

RQ1: To what extent can we improve the energy efficiency of an application by statically replacing Java collection implementations?

RQ2: Are the recommendations device-independent?

To answer **RQ1** and **RQ2**, CT+ analyzed the source code of 17 software systems across 7 devices, making energy saving recommendations for 13 of them. By analyzing the recommendations made by CT+ and their effect on the energy consumption of these systems, we concluded that some very popular collection implementations, such as `ArrayList`, `Hashtable` and `HashMap`, have poor energy efficiency.

RQ3: How much does the workload size impact the energy efficiency of a Java collection implementation?

RQ4: Are the recommendations profile-independent?

To answer **RQ3** and **RQ4**, we created 6 different energy profiles, simulating distinct kinds of workloads an application may be subject to. We applied the recommendations made by CT+ to 6 systems of the DaCapo benchmark suite. The results show that these different profiles behave very differently, depending on the circumstances. The difference goes up to

6.18× more energy saved when comparing the best performing profile with the worst one for the same software system.

Overall, 2295 recommendations that impacted the energy consumption were made across 17 software systems, 12 targeting a desktop environment, 2 targeting a mobile environment, and 3 that work in both scenarios, for a total of 64 modified versions. Most of these systems were non-trivial with thousands of lines of code (LoC), such as BIOJAVA with 914kLoC, CASSANDRA with 466kLoC, and TOMCAT with 433kLoC. With no prior knowledge of the application domains or the system implementations, CT+ made positive recommendations for 13 out of the 17 systems. It was possible to reduce the energy consumption of software systems up to 16.34% on a desktop environment and 14.78% on a mobile environment, just by replacing collection implementations. For a small number of modified versions (2 out of 64), recommendations made by CT+ degraded the energy efficiency, up to 1.21%.

The results of our studies highlight the need to re-assess the adoption of some widely popular, poorly-optimized collection implementations from the Java Collections Framework, such as `ArrayList`, `Hashtable`, and `HashMap`. Recommendations to replace uses of these collection implementations by more efficient alternatives were common in our evaluation. In addition, there was not a single case where `HashMap` was was recommended, and just three cases for `Hashtable`, with data suggesting that `Hashtable` is becoming less used by developers. Furthermore, 89% of the recommendations made by CT+ suggested the use of collection implementations not from the JCF. The data related to this work can be found at https://energycollections.github.io/.

This work is an extended and improved version of a previously-published paper (Oliveira et al. 2019). We improved the original work in a number of different ways: (i) Two new research questions about the energy behavior of Java collections were answered (i.e., **RQ3** and **RQ4**); (ii) Brand new experiment using a different methodology aiming to evaluate the impact of using different strategies to build energy profiles was added; (iii) Five new, mature software systems were included in our pool. (iv) More in-depth analysis of collection usage was done, in particular, the impact of profile creation strategies and workloads on the recommendations and how to make profile creation time-efficient; and (v) A new mobile device and desktop machine were included in the original study.

This article is structured as follows: Section 2 briefly describes the importance of Java Collections; Section 3 presents the related work, with a particular focus on empirical studies about energy consumption and Java Collections recommendation tools; Section 4 introduces our approach to help develop more energy efficient software systems; Section 5 demonstrated how we used our proposed approach to develop a recommendation tool, CT+. This tool focus on optimizing applications to use the most energy efficient collections implementation in Java; Section 6 displays the results from our experiments analyzing the energy consumption of different devices and energy profiles; Section 7 report our findings about the energy consumption of Java Collections; Section 8 presents the threats to validity; and Section 9 shows the conclusion of this article.

2 Java Collections

Collections are widely used by developers, in both mobile and desktop environments. They provide easy access to reliable implementations that can reduce the complexity of developing applications. Java's collections in particular are usually subdivided in three different APIs: `Lists`, `Maps`, and `Sets`. These categories are divergent 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's API has multiple implementations. This is expected since there is a number of different algorithms and data structures that can implement the abstract concept of lists, sets and maps. Examples include using dynamically-allocated list nodes and making the list doubly-linked, as in `LinkedList`, or a resizable array, as in `ArrayList`. Although both are implemented differently, they still respect the set of rules of a list and implement the same interface, `List`. As a consequence, it is often possible to change the list implementation being used in a given context by modifying a single line of code, i.e., the line where the collection implementation class is instantiated.

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 the use of a thread-safe collection implementation with a thread-unsafe one. Conversely, although it is possible to replace a thread-unsafe collection implementation using a thread-safe one, this is not efficient in practice (Pinto et al. 2016).

There are many different ways a collection can be implemented, and these diverse implementations 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 (Pinto et al. 2016; Hasan et al. 2016; Costa et al. 2017) 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[5], Apache Commons Collections[6], and Eclipse Collections[7].

To get a glimpse at the usage of these alternative implementations in Java projects, in April 2020 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 query results showed that these collections are in widespread use, with 1,276,939 cases for Apache Common Collections and 537,956 for Eclipse Collections.

As expected, JCF collections, both thread-safe and thread-unsafe, are also in widespread use. To investigate the adoption of JCF, we conducted another simple query but, differently from Apache and Eclipse Collections, JCF does not have a package exclusively for collections. As a consequence, the query for its implementations collections were executed individually, using the full package name (e.g., `java.util.ArrayList`). The results suggest that thread-unsafe collections are used more often than thread-safe collections by a fair margin. For example, one the one hand, the most widely used thread-unsafe collection is `ArrayList`, a `List` implementation, with 38,642,021 occurrences in our query. On the other hand, the most widely used thread-safe collection is `Vector`, another `List` implementation, with 5,526,922 occurrences.

In our previous study (Oliveira et al. 2019), we conducted a similar query analyzing collection usage in Github projects as of January of 2019. As shown in Table 1, overall, there was an increase in collection usage for all collection types.

[5] Java Collections Framework webiste: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/
[6] Apache Commons Collections website: https://commons.apache.org/proper/commons-collections/
[7] Eclipse Collections website: https://www.eclipse.org/collections/

Table 1 Adoption of collections across Github Java projects

Implementation	Jan. 2019	Apr. 2020	Growth
Thread unsafe collections			
`ArrayList`	35,278,092	38,642,021	9.54%
`HashMap`	16,602,391	19,418,572	16.96%
`HashSet`	6,470,505	8,104,539	25.25%
`LinkedList`	3,763,660	4,577,164	21.61%
`LinkedHashMap`	1,470,500	1,953,047	32.82%
`TreeMap`	1,122,886	1,370,672	22.07%
`TreeSet`	950,890	1,174,078	23.47%
`LinkedHashSet`	689,397	944,604	37.02%
Sum of thread unsafe	**66,348,321**	**76,184,697**	**14.83**%
Thread safe collections			
`Vector`	4,731,762	5,526,922	16.80%
`Hashtable`	1,994,173	2,084,408	4.52%
`ConcurrentHashMap`	1,119,704	1,575,483	40.71%
`CopyOnWriteArrayList`	237,541	310,188	30.58%
`CopyOnWriteArraySet`	70,680	94,507	33.71%
`ConcurrentSkipListMap`	39,012	52,394	34.30%
`ConcurrentSkipListSet`	26,826	36,671	36.70%
Sum of thread safe	**8,219,698**	**9,680,573**	**17.77**%
`org.apache.commons.collections`	1,022,778	1,276,939	24.85%
`org.eclipse.collections`	466,394	537,956	15.34%

All implementations came from the package `java.util`

The table explicitly separates thread-safe and thread-unsafe JCF collection implementations, since they represent the vast majority of the collection implementations. Nevertheless, the Apache and Eclipse Collections both have thread-safe and thread-unsafe implementations.

Our queries results show that the average increase in the utilization of the collection implementations between January 2019 and April 2020 was 14.83% for thread-unsafe collections and 17.77% for thread-safe. The only two collections with a growth rate of less than 10% were `ArrayList` (9.54% and 3,363,929 new occurrences) and `Hashtable` (4.52% and 90,235 new occurrences).

Because of the extensive adoption of `ArrayList`, the growth rate naturally tends to slow down, since most projects already use it. `Hashtable`, on the other hand, appears to have fallen from grace, with developers opting to use other solutions when they need to use a thread-safe map implementation, like `ConcurrentHashMap` (40.71% growth rate and 455,779 new occurrences). The data from our first query shows `Hashtable` with 78% more occurrences than `ConcurrentHashMap`. However, in our last query, that difference has been reduced to 32%. If the growth rate continues, by year 2021 `ConcurrentHashMap` will have more occurrences than `Hashtable`.

3 Related Work

Energy profiling research has been conducted in different contexts, including embedded systems (Šimunić et al. 2000), cloud computing (Chen et al. 2011), concurrent programming primitives (Pinto et al. 2014b; Lima et al. 2016), neural networks and models (Romansky et al. 2017). These studies share a common finding: simple changes can reduce energy consumption considerably. However, most of these studies do not provide tool support for developers. If interested, developers are still required to have: (1) the infrastructure (software and, eventually, hardware) to conduct the experiments, and (2) in-depth knowledge of low-level implementation details. As a result, non-specialist developers have little chance to apply the findings in real-world scenarios. In contrast, our approach is focused on non-expert developers. They do not have the knowledge neither, time, or tools to understand the energy impact of energy variation hotspots, but still want to reduce energy consumption. With an appropriate design and implementation (Sections 4 and 5), we believe that our approach can be reused and useful at scale.

In this article, we introduce a general idea to save energy during software development and instantiate it in a tool called CT+. We take this into account by organizing related work in terms of empirical studies (which create knowledge about energy efficiency) and recommendation tools (which apply that knowledge).

Empirical studies In recent years, researchers have empirically analyzed several different aspects of energy consumption on software engineering. Rocha et al. (2019) took a look at the energy behavior of I/O APIs on 22 Java benchmarks and 3 macro-benchmarks. They showed that there is not a single API that can be used on every application independently. Another important aspect of their work was that that small modifications on these APIs can result in a better energy performance by already optimized applications. Georgiou and Spinellis (2020) investigated the energy consumption impact of seven different programming languages on inter-process communication, finding out that the implementations on Javascript and Go usually dare the most energy efficient. Duarte et al. (2019) developed a framework based on model analysis to study the energy consumption of software systems and then evaluated it experimentally. Among the different usages of their framework, one is to detect refactoring points that could be changed to reduce energy consumption.

Another way to look into the ways developers can save energy is try optimize their energy choices even before the system development starts, that is, at the design phase. Sahin et al. (2012) made an investigation about the energy impact of 15 different design patterns. Their results shows that design patterns could have a high effect on the energy consumption, going from a pattern that consumes 10 times more energy than the original code to another one that only consumes half the energy. In a similar fashion, Cruz and Abreu (2017) analyzed the energy impact of code smells on Android applications. On their work, they present empirical evidence that these anti-patterns increase the energy consumption of mobile applications and should be avoid by developers to help create more energy efficient apps. Lyu et al. (2017) analyzed the usage of local database requests in Android applications, founding out that the most expensive operations are database initialization and write operations. These operations are often used in loops and developers could design them to be bundled to reduce their energy consumption. Chowdhury et al. (2019a) studied the impact of different strategies to optimize the energy consumption of the Model-View-Controller architectural pattern (MVC). Two strategies were used to optimize how these types of applications handle data influx: (i) bundling the data updates or (ii) using only the most recent one. These strategies were used to create new versions of the applications. With changes that are almost

imperceptible to the human eye and do not impact the user experience, they showed that it is possible to reduce the energy consumption of a MVC application up to 36%.

More specifically, some works have dealt only with recommending more energy-efficient collections. Hasan et al. (2016) 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. (2016) 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 `Map`. Saborido et al. (2018) 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. (2014) 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. (2018) implemented an energy-aware tool called jStanley, aiming to recommend the best collection implementation among several over the Java Collections Framework. jStanley was implemented as an Eclipse plugin and worked using experimental results from prior work by the same authors (Pereira et al. 2016). It does not account for the impact of loops and it works exclusively in a mobile environment. Furthermore, it does not account for thread-safety.

As stated before, this work is an extended and improved version of a previously-published paper analyzing the energy consumption of Java collections (Oliveira et al. 2019). With this extension we aim to improve the understanding of the energy behavior of Java collections.

4 Overview of the Proposed Approach

In this section we propose a novel approach to help developers create more energy efficient applications. This approach can be used for general-purpose development, not being restricted to a specific scenario, development environment, device, or application. The proposed approach is organized in three phases: (i) creation of the energy profiles, (ii) collection usage analysis, and (iii) recommendation of source code modifications that have the potential to reduce energy consumption. Figure 1 provides an overview of our approach.

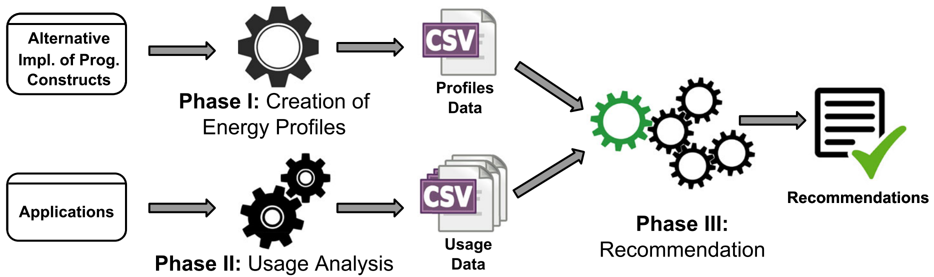


Fig. 1 An overview of our approach. Phase I is application-independent, Phase II is device-independent, and Phase III uses the energy profile and the information about the system under analysis

In this section we present a high level overview, with the three phases detailed next. In Section 5 we present our instantiation of this approach in the CT+ tool, dealing with the optimization of Java collections.

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 (Hasan et al. 2016). *The energy profile can be seen as a set of numerical values representing the energy cost of a specific construct, making it possible to compare the energy efficiency of similar constructs under the same circumstances.* The main insight of using energy profiles is that it is possible to order interchangeable pieces of software by their energy consumption, without actually quantifying their energy consumption. This idea can be explored in diverse situations. For example, previous work (Linares-Vásquez et al. 2015; Wan et al. 2017; Linares-Vásquez et al. 2018) 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 5 we define energy profiles in a more precise manner for the specific context of collection implementations. The energy profiles created in Phase I are used as input to make the recommendations 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. 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. However, the static approach 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.

Dynamic approaches make it possible to estimate more precisely how intensive energy variation points will be used in realistic scenarios. Notwithstanding, the precision of dynamic approaches depends heavily on the employed workload, specially for complex software systems that make intensive use of multiple resources (CPU, disk, wired network, wireless network, etc.). Dynamic approaches could benefit more UI-focused applications, as these are harder to exercise with static analysis. It could also be useful for cases where part of the logic is outsourced or there's a heavy dependency on non-deterministic factors (e.g., GPS locations or temperature).

The output of this phase is heavily dependent on a number of different factors. Examples may include the kind of construct, line of code where the construct is used, the number of times it is instantiated or invoked, the 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, taking the data produced by Phases I and II as input. Different formulae can be employed in this phase. A straightforward approach is to linearly combine the energy profiles (created in Phase I) with the frequency of use (through the analysis made in Phase II) of the 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. Nonetheless, as with the previous phase, there is ample opportunity to explore different solutions to combine these two pieces of information.

These three phrases summarize our approach to help developers save energy while developing software systems. To be able to offer guidance in as many scenarios as possible, they are open to a number of different instantiations. Phase I is application independent while Phase II is execution-environment (device, operating system, runtime system) independent, leaving the possibility for the developer to leverage multiple implementations of each phase.

5 Instantiation for Java Collections

In this section, we describe how we have instantiated the proposed approach to work with Java collections. We developed a tool named CT+ that implements the three phases explained in the previous section.

CT+ analyzes Java programs in desktop and mobile environments, recommending collections that could potentially reduce energy consumption. We organize the presentation of the instantiation in terms of the three phases introduced by Section 4.

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.

The three analyzed collections have differences among them. For example, on lists it is possible to insert or to 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 2 presents a summary of the operations we analyze to build the energy profiles. Although there are other possible operations (e.g., `List.removeAll()`), they are not used on this study.

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 implementation, e.g., `ArrayList`, `HashMap`, etc., T is the type of the collection, with $T \in \{List, Set, Map\}$, S is the thread-safety of the collection, with $S \in \{ThreadSafe, NotThreadSafe\}$, and o_i, with $1 \leq i \leq n$ represent the operations of the collection implementation. The following tuple is an example of how `Vector` could be represented:

$$C_v = (Vector, List, ThreadSafe, \texttt{insert.start(e)}, \texttt{insert.end(e)}, ..., o_n)$$

In the previous example, `insert.start(e)` and `insert.end(e)` indicate that these are operations that insert an element in the beginning and at the end of the list, respectively.

The energy profile for a collection implementation is a tuple whose elements are numbers, e.g., energy consumption in joules, that can be used to compare the energy cost of the same operations for different collection implementations under the same execution environment. The energy profile of a collection implementation C is produced by a profiler, a function that, in a given execution environment and under a set of workloads, produces an energy profile:

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

where env abstractly represents the execution environment in which the profiler is running (machine, operating system, JVM version). w_i, $1 \leq i \leq n$, is the workload for the operation o_i, defined by the function $workload(N, o_i)$, which produces a workload given the name of a collection implementation and one of its operations. Finally, e_i, with $1 \leq i \leq n$, is the energy consumption value for o_i.

Two energy profiles P_a and P_b for two collection implementations C_a and C_b, respectively, can be compared as long as three constraints are satisfied: $C_a.T = C_b.T$, $C_a.S = C_b.S$, and $P_a.env = P_b.env$. In other words, we cannot, for example, compare energy

Table 2 Operations used on each collection

Operation	Types
Lists	
insertions	start, middle, end, and default
iterations	random, iterator, and loop
removals	object, start, middle, end, and default
Maps	
insertions	default
iteration	iterator and loop
removal	default
Sets	
insertions	default
iteration	loop
removal	default

profiles for a list and a set. In the same vein, it is not possible to compare a profile for a thread-safe implementation and one for a non-thread-safe implementation, nor profiles obtained in two different execution environments. We assume that collection implementations whose T and S elements are the same are, from an implementation standpoint, functionally equivalent. For example, adding an element to an `ArrayList` is functionally equivalent to adding that element to a `LinkedList`, although this would not be true for a `Vector`, since the latter is thread-safe. This is true in practice for the vast majority of the collection implementations in the JCF, with very few exceptions (e.g., `WeakHashMap`).

Table 3 lists all implementations analyzed in this work. 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`.

We build energy profiles by running micro-benchmarks applied to the operations of each collection implementation. Micro-benchmarks are a set of instructions to measure the energy consumption of a specific piece of code by exercising it repeatedly. In our case, a micro-benchmark executes our selected operations a predetermined number of times for every implementation. For example, to collect the energy data, we execute 60,000 times the `ArrayList`'s micro-benchmark to measure the `add(start)` method in one of our devices. That predetermined number is energy-profile dependent and we discuss this in more depth in Section 6.2.2.

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(e)` and `remove.start(e)`). 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.

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 on 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. (2007) for Java performance evaluation. The energy profiles for each collection implementation are used in 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 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. Each time an operation appears on the source code, CT+ adds it to the file containing all operations used by that software system. One operation can be counted more than once, depending on its context. For example, the operation "`collection.bar()`" can be called from the method "`foo()`" in different parts of the code; one invocation of "`collection.bar()`" might be inside a loop while another one may not be involved in loops. Each of these operations is counted separately by CT+.

Table 3 The selected implementations to be used in the experiments

Thread Safety	Implementations	
Lists		
Safe	Vector	CopyOnWriteArrayList
	SynchronizedArrayList	SynchronizedList
	SyncronizedFastList	
Unsafe	ArrayList	LinkedList
	FastList	CursorableLinkedList
	NodeCachingLinkedList	TreeList
Maps		
Safe	Hashtable	ConcurrentHashMap
	ConcurrentSkipListMap	SynchronizedHashMap
	SynchronizedLinkedHashMap	SynchronizedTreeMap
	SynchronizedWeakHashMap	ConcurrentHashMap(EC)
	SynchronizedUnifiedMap	StaticBucketMap
Unsafe	HashMap	LinkedHashMap
	TreeMap	UnifiedMap
	HashedMap	
Sets		
Safe	ConcurrentSkipListSet	CopyWriteArraySet
	SetConcurrentHashMap	SynchronizedHashSet
	SynchronizedLinkedHashSet	SynchronizedTreeSet
	SynchornizedTreeSortedSet	SyncronizedUnifiedSet
Unsafe	HashSet	LinkedHashSet
	TreeSet	TreeSortedSet
	UnifiedSet	

We employed three different sources: Java Collections Framework, Eclipse Collections and Apache Commons Collections

In this phase, among other pieces of information, for each invocation of a collection operation, we collect: class name, collection type, concrete type, calling method, name of the field storing the collection implementation object, invoked collection operation, line of code, whether the invocation appears within a loop, and whether the invocation is performed from within a recursive method.

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.,* Rodrigues et al. 2014), these works (1) do not cover many practical loop usage scenarios for languages such as Java, where arrays are 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 CT+ makes the recommendations based on three different factors about each collection implementation, for each target system: (i) the energy profile information for that collection implementation; (ii) data about occurrences of the collection operations within the target system; and (iii) whether those occurrences appear within loops or not.

More specifically, for each object c that is an instance of a collection implementation C and each path in the program call graph where an operation on c is invoked, considering every collection implementation C' such that $C.T = C'.T$, $C.S = C'.S$, and the profiles for C and C' were built on the same execution environment, CT+ calculates (1) the energy cost of the operations outside loops, (2) the energy of operations inside loops, and then combines these two pieces of information to calculate the energy factor EF according to (3), which combines (1) and (2).

$$E^{\neg L}(C', c) = \sum_{i=1}^{n} e_i * NL(c.o_i) \tag{1}$$

In this formula, n is the number of operations in C, e_i is the ith 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.

$$E^{L}(C', c) = \sum_{j=1}^{n} \sum_{d=1}^{m} e_j * ((L_d(c.o_j) + 1)^{d+1} - 1) \tag{2}$$

Differently from (1), here 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 (i.e., $L_d(c.o_j) = 0$) get canceled out.

$$EF(C', c) = E^{\neg L}(C', c) + E^{L}(C', c) \tag{3}$$

The first factor of each summation is originated from the energy profile created in Phase I (i.e., e_i and e_j) while the second factor comes from data collected in Phase II (i.e., NL and L_d).

CT+ makes its recommendation based on the energy factors of the collection implementations. It provides as output an ordered list of collection implementations with better energy footprint than the original collection. Once the implementation is chosen, CT+ can automatically refactor the application, within the context of the recommendation, to use the first element of the ordered list.

Implementation To collect the energy consumption of the different devices, two profilers were created, one for the desktop enviroment and one for the mobile enviroment.

The **Desktop Profiler** is a simple Java program that uses the jRAPL (Liu et al. 2015) library to measure energy consumption on desktop applications. It works on Intel architectures (starting with Sandy Bridge, released in 2011). 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 (released in 2014) or later.

In both environments, the whole application energy consumption data is collected. That means that we analyze how much energy was consumed by the entire application during its execution while running different collection implementations. On mobile devices, the Android Power Profile is used to isolate the energy consumption of the application. On desktop devices, RAPL uses machine specific registers (MSR) to read the stored the energy consumption. The energy cost of an operation is calculated using the difference between the energy data on the register before and after it's execution. jRAPL includes the execution cost of the underlying system.

The analysis and recommendation aspects of CT+ are based on WALA[8], 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, making it possible to recommend collections satisfying the same constraints of thread-safety. This is also useful to support recommendations that account for collection objects being passed as method arguments.

The whole CT+ project was developed in Java and has 23kLoc, comprising two energy profilers, the analysis tool, the recommendation tool, and the auxiliary dashboard application for mobile experiments. The individual link for the repositories can be found at https://energycollections.github.io/. To help developers and researchers, we also made available in our website a cookbook guiding the developer to use it together with DaCapo.

6 Evaluation

In this section, we present the evaluation of the proposed approach. We applied CT+ to a number of software systems, running on multiple execution environments. The main objective of this evaluation is to compare the energy consumption of the original versions of these systems with modified versions where the recommendations made by CT+ were applied.

The evaluation is divided in two parts. In Section 6.1, we examine the efficiency of CT+'s recommendations when considering different execution environments. Then, in Section 6.2 we analyze the impact on energy efficiency of using six different strategies to build energy profiles. For this second part, we run all the experiments on a laptop.

When executing the benchmarks on the desktop devices, two different versions of the DaCapo suite (Blackburn et al. 2006) were used: version 9.12 and development version 19.07. This was the case because since January 15, 2020, Maven's Central Repository no longer supports communication over HTTP[9]. Because of that, the older versions of DaCapo that required the usage of Java Development Kit (JDK) up to version 6 do not work anymore. Using these older versions on newer devices can be challenging, as it requires in-depth knowledge about DaCapo as well as rewriting potentially dozens of configuration files. While executing version 19.07, JDK 8 was used. This newer JDK version is supported by current versions of DaCapo and complies with Maven's new policy. The study presented in Section 6.1 uses version 9.12 of DaCapo for two devices and version 19.07 for one. The study presented in Section 6.2 uses exclusively the latest version.

[8]WALA website: http://wala.sourceforge.net/wiki/index.php/Main_Page

[9]Maven's new policy: https://central.sonatype.org/articles/2019/Apr/30/http-access-to-repo1mavenorg-and-repomavenapacheorg-is-being-deprecated/

Although execution time is not, in general, a proxy to energy consumption (Hao et al. 2013; Li et al. 2014a; Pang et al. 2016; Chowdhury et al. 2019b), sometimes it can be a good approximation that is more convenient to use. To verify if there was a correlation between the execution time and the energy consumption, we calculate the Spearman Correlation between execution time and energy consumption on the device used in Section 6.2, for the systems where CT+ recommendations made a statistically significant impact on the energy consumption. We found out that there was a statistically significant difference for 62.86% of the cases. This result suggests that analyzing the energy consumption separately from execution time may still be the more appropriate approach, since it was not a good approximation for energy in more than 1/3 of the cases.

6.1 Analyzing Different Devices

This section describes our experimental environment and results from the study using CT+ to analyze the energy efficiency of Java collections on different devices. Overall, seven different devices were used in this experiment: a high-end server, two notebooks, three smartphones, and a tablet.

The remainder of this section is organized as follows. Section 6.1.1 lists the research questions we aim to answer; Section 6.1.2 describes the methodology of this study, including the seven aforementioned devices and the target software systems of the study; and Section 6.1.3 presents the results.

6.1.1 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 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, we aim to answer the following research questions (RQs):

RQ1: To what extent can we improve the energy efficiency of an application by statically replacing Java collection implementations?

RQ2: Are the recommendations device-independent?

6.1.2 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.

Desktop environment CT+ was executed across three different machines on the desktop environment, two notebooks and a high-end server. We labeled the notebooks as **dell** (Dell Inspiron 7000) and **asus** (Asus X555U), and the server as **server**. **dell** has an Intel Core i7-7500U processor with two 2.7GHz cores with four threads, and 16GB of RAM. **asus** has an Intel Core i5-6200U processor with two 2.2GHz physical cores, with four threads and 8GB

Table 4 Machines used on the study about devices

Device	Alias	RAM	Chipset	CPU (GHz)	Battery	Age
Desktop						
Inspiron 7000	**dell**	16GB	i7-7500U	2-core 2.70	N/A	N/A
Server	**server**	256GB	Xeon E5-2660	20-core 2.2	N/A	N/A
Asus X555U	**asus**	8GB	i5-6200U	2-core 2.80	N/A	N/A
Mobile						
Samsung J7	**J7**	1.5GB	Exynos 7580	8-core 1.5	3000 mAh	3
Samsung S8	**S8**	4GB	Exynos 8895	8-core 2.3'1.7	3000 mAh	1
Motorola G2	**G2**	1GB	Snapdragon 400	4-core 1.2	2070 mAh	4
Samsung Tab4	**Tab4**	1.5GB	Cortex-A7	4-core 1.2	4000 mAh	6

Age shows how old the device was when we executed the experiments (in years)

of RAM. **server** has two-node Intel Xeon E5-2660 v2 processor with 20 2.20GHz physical cores (10 per node) and 20 "virtual" cores[10], and 256GB of RAM. In the experiments, we always execute benchmarks on **dell** and **asus** while they are connected to the power outlet, since we are using it as a desktop machine.

Mobile environment We executed our tool on three smartphones and a tablet: Samsung Galaxy J7 (**J7**), Samsung Galaxy S8 (**S8**), Motorola G2 (**G2**), and Samsung Galaxy Tab 4 (**Tab4**). Table 4 presents a summary of the devices used in this study in both environments. All experiments were executed while these mobile devices were in discharge mode (not connected to a power outlet), their more typical usage scenario. For all cases, battery charges ranged between 100% and 80%. The latter restriction aims to reduce influence of dynamic voltage and frequency scaling on the measurements. We also report the age of each mobile device at the time the experiments were run, since older batteries tend to exhibit more erratic discharge patterns (Lee et al. 2015). Section 8.1 discusses this further.

When creating the energy-profiles, we chose to use the same methodology as previous work (Hasan et al. 2016; Oliveira et al. 2017), although we leverage a larger number of collection implementations (Table 3). 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. Before collecting the energy data samples, we performed a warmup execution. In the warmup, we executed up to 10% of our workload. By doing this, we minimized JIT noise on the measurements (Barrett et al. 2017a).

As previously explained, for this experiment, two different versions of DaCapo were used. To execute the benchmarks on **dell** and **server**, we used DaCapo version 9.12, and on **asus**, we used version 19.07. Because different versions of DaCapo were used, there are also two distinct versions of TOMCAT. DaCapo version 9.12 uses TOMCAT 6.0.20 (TOMCAT V6 for short) while version 19.07 uses TOMCAT 9.0.2 (TOMCAT V9 for short).

On **dell**, we analyzed seven desktop-based software systems: BARBECUE, BATTLECRY, JODATIME, TOMCAT V6, TWFBPLAYER, XALAN, and XISEMELE; two mobile-based software systems: FASTSEARCH and PASSWORDGEN; and three systems that work on both

[10]Summary about hyper-threading: https://en.wikipedia.org/wiki/Hyper-threading

Table 5 Software systems used on the study about devices and where they were executed

Device	Selected software systems			
Environment: Desktop				
dell	TOMCAT V6	XALAN	BARBECUE	BATTLECRY
	JODATIME	TWFBPLAYER	XISEMELE	
	COMMONS MATH 3.4	GOOGLE GSON	XSTREAM	
asus	TOMCAT V9	BIOJAVA	CASSANDRA	GRAPHCHI
	KAFKA	ZXING		
server	TOMCAT V6	XALAN		
Environment: Mobile				
all	FASTSEARCH	PASSWORDGEN		
	COMMONS MATH 3.4	GOOGLE GSON	XSTREAM	

environments: APACHE COMMONS MATH 3.4 (COMMONS MATH for short), GOOGLE GSON, and XSTREAM: These systems were employed in related work on energy profilling (Sahin et al. 2014; Pinto et al. 2016; Hasan et al. 2016; Pereira et al. 2018), and their workloads are available for replication purposes. For **server**, we only ran TOMCAT V6 and XALAN since these are applications one would expect to execute on a high-end server machine.

For **asus**, we executed six systems: BIOJAVA, CASSANDRA, GRAPHCHI, KAFKA, TOMCAT V9 and ZXING. XALAN was not executed on **asus** because the project seems to be abandoned. As of April 2020, the last update on GitHub's page was on June 2001[11]. Table 5 summarizes the software systems used in this study and the devices in which they were analyzed.

It is possible to tune the workload size on DaCapo's benchmarks and different benchmarks have distinct options for their workloads. TOMCAT is a particular application among DaCapo as it has four different workload sizes (SMALL, DEFAULT, LARGE, and HUGE). XALAN has three different workload sizes (SMALL, DEFAULT, and LARGE). The other applications used on this study only have one (DEFAULT). The advice of DaCapo developers is to use the biggest option available[12].

Each system has a specific workload and routine to follow with the objective of exercising different aspects of their source code. As an example, executing BIOJAVA creates 10 physico-chemical properties of different-sized protein sequences, TOMCAT runs a number of web applications, and GRAPHCHI uses the Netflix Prize dataset (Bennett et al. 2007b) to drive it's engine. These routines are all selected and curated by DaCapo developers. For TOMCAT V6 and XALAN on the desktop development machines, we used the same workloads sizes (i.e., LARGE) while on **asus**, TOMCAT V9 was executed with two different workloads sizes (i.e., LARGE and HUGE). The number of threads also varied between devices: forty on **server** and four on **dell** and **asus**.

The workloads of systems outside of DaCapo suite were done in more individualized manner. For four systems (BARBECUE, JODATIME, TWFBPLAYER, XISEMELE) we used

[11]Xalan project: https://github.com/apache/xalan-java

[12]DaCapo website: http://dacapobench.sourceforge.net/benchmarks.html

unitary tests, following the same methodology as previous work (Pereira et al. 2018). 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 utility programs that work like functions, their workloads consisted of executing their `main` methods (e.g., generating passwords).

For TOMCAT V6, we could not recommend any implementation from the Eclipse Collections library. This happened because version 9.12 of DaCapo requires the use of Java Development Kit (JDK) up to version 6 to ensure the correct operation of its benchmarks. Unfortunately, the current version of Eclipse Collections is incompatible with JDK 6. For this particular benchmark, our tool still makes recommendations with JCF and Apache Commons Collections.

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 (Liu et al. 2015) 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 (Shapiro and Wilf 1965), we used the Wilcoxon-Mann-Whitney test (Wilks 2011) 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 (Cliff 1993) as a measure of effect size. Wilcoxon-Mann-Whitney test and Cliff's Delta are non-parametric tests.

6.1.3 Study results

We present the results in terms of the desktop and mobile environments. For each one, we first present 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 6.1.2 most executions had a designed workload based on the time necessary to execute them. The specific amount of time each system took to be executed can be found at https://energycollections.github.io/. In this experiment, across all devices, a total of 584 recommendations were made.

Desktop environment Table 6 summarizes the energy consumption for the desktop environment. The most important column of the table is **Improvement**, which shows how much more energy the original version consumed, when compared to a modified version where CT+'s recommendations have been applied. 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 than the original versions. For five of them, TWFBPLAYER and XISEMELE on **dell**, TOMCAT V9 using the HUGE workload, KAFKA, and CASSANDRA on **asus**, the difference between original and modified versions was not statistically

Table 6 Results for the desktop environment

System	Improvement	p-value	Mean(J)	Stdev	Effect size
Development Machine: **dell**					
Barbecue	4.38%	7.0^{-4}	56.17	2.70	0.50
			53.71	2.53	
Battlecry	2.78%	1.5^{-3}	67.95	2.67	0.48
			66.06	3.18	
Google Gson	0.72%	8.0^{-5}	29.93	0.22	0.57
			29.72	0.16	
Commons Math	1.04%	6.3^{-12}	48.93	0.29	0.90
			48.43	0.15	
JodaTime	6.66%	$< 2.2^{-16}$	123.02	2.42	0.94
			114.83	3.50	
Tomcat v6	3.96%	$< 2.2^{-16}$	32.77	1.02	0.86
			31.47	0.41	
Xalan	4.77%	$< 2.2^{-16}$	107.04	0.19	1
			101.93	0.15	
Xstream	2.52%	3.12^{-13}	59.97	0.52	0.94
			58.45	0.49	
Development Machine: **asus**					
Biojava	0.60%	2.20^{-16}	193.69	0.11	1
			192.53	0.17	
Graphchi	10.17%	5.79^{-13}	10.90	0.35	0.94
			9.80	0.42	
Tomcat v9-LARGE	1.04%	2.53^{-3}	74.93	1.37	0.31
			74.16	1.78	
Zxing	5.84%	2.20^{-16}	85.82	0.36	1
			80.80	0.41	
Development Machine: **server**					
Tomcat v6	4.12%	$< 2.2^{-16}$	89.21	2.99	0.66
			85.54	2.37	
Xalan	5.49%	$< 2.2^{-16}$	242.29	4.4	0.86
			228.98	7.02	

Energy results are red for the original versions and green for the modified versions. Energy measured in Joules

significant. In the case of TOMCAT V9 using the LARGE workload, there was a significant difference with a small effect size. Notwithstanding, for the remaining systems, the difference is statistically significant and the effect size is large.

According to Romano et al. (2006), effect size as measured by Cliff's Delta has four different categories: large ($\geq$ 0.474), medium (between 0.474 and 0.33), small (between 0.33 and 0.147), and negligible ($\leq$ 0.147). For example, on XALAN in the **dell** 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 **dell** machine, JODATIME exhibited the greatest improvement, with the modified version consuming 6.66% less energy than the original one. To make it easier for the reader to focus on the relevant data, this section focused 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. The data about the non statistically significant results can be found at https://energycollections.github.io/.

The two software systems that were executed in the **dell** and **server** machines, XALAN and TOMCAT V6, exhibited positive results in both scenarios. For XALAN, the modified version consumed less 4.77% and 5.49% energy than the original version in the **dell** and **server** machines, respectively. For TOMCAT V6, the differences were of 3.96% and 4.12%, respectively. We found that systems running on **server** consumed more than twice the energy they consumed on **dell**, 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.

Meanwhile, TOMCAT V9 exhibited a different behavior on **asus**. Using the workload size HUGE, the recommendations made by CT+ did not result in energy reduction. Nevertheless, while using the same workload as in **dell** and **server**, there was a reduction of 1.04%, an improvement that represents less than 25% the improvement made on the other two devices. Since on **asus** we executed a newer version of TOMCAT, smaller improvements were expected, since developers behind the newer versions of TOMCAT are likely to be more mindful of the collection implementations being used.

For all the other systems executed on **asus**, CT+ recommendations resulted in a reduction in energy consumption with a large effect size. In particular, GRAPHCHI was the system with the most significant reduction in energy consumption among all systems across all desktop development machines (10.17% of improvement).

Tables 7, 8 and 9 summarize the recommendations for each application on **dell**, **asus**, and **server**, respectively. The first column lists the names of the target systems for the desktop environment. The second column presents the names of collection implementations used in these systems, whereas the third column indicates the collection implementations that CT+ recommended using instead. Finally, the fourth column displays the number of times CT+ recommended the one in the third column as a replacement to the corresponding collection implementation in the second column. Overall, on the desktop environment, CT+ made and applied 724 recommendations lead to statistically significant results.

In **dell** and **server** machines XALAN had a significant number of `Hashtable` implementations changed to `ConcurrentHashMap(EC)` (48 and 49 times on **dell** and **server**, respectively). For both machines, 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 V6 recommendations differed across these two machines. On **dell**, the tool made thirteen 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 1, these are widely-used collections. We reiterate that Eclipse Collections could not be recommended for TOMCAT V6 (Section 6.1.2).

Table 7 Recommended collection implementations for the **dell** machine and how many times they were recommended.

System	Original	Recommended	# of times
Development Machine: **dell**			
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	ConcurrentHashMap(EC)	1
JodaTime	ArrayList	FastList	8
	HashMap	HashedMap	7
	ConcurrentHashMap	ConcurrentHashMap(EC)	1
Tomcat v6	Hashtable	ConcurrentHashMap	6
	HashMap	HashedMap	4
	Hashtable	StaticBucketMap	2
	Vector	Synchronized LinkedList	1
Xalan	Hashtable	ConcurrentHashMap(EC)	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

Table 8 Recommended collections for **asus**

System	Original	Recommended	# of times
Development Machine: **asus**			
Biojava	HashMap	HashedMap	100
	ArrayList	FastList	37
	LinkedList	ArrayList	2
	TreeSet	TreeSortedSet	2
	ArrayList	NodeCachingLinkedList	1
	HashSet	UnifiedSet	1
	Hashtable	ConcurrentHashMap	1
	Vector	SynchronizedArrayList	1
Graphchi	HashMap	HashedMap	3
	ArrayList	FastList	1
Tomcat v9	HashMap	HashedMap	47
	ArrayList	FastList	9
	ConcurrentHashMap	ConcurrentHashMap(EC)	8
	CopyOnWriteArrayList	SynchronizedArrayList	8
	ConcurrentHashMap	SynchronizedHashMap	5
	ConcurrentHashMap	Hashtable	3
	Hashtable	ConcurrentHashMap(EC)	4
	Hashtable	SynchronizedHashMap	4
	LinkedList	TreeList	2
	LinkedList	FastList	1
	TreeSet	TreeSortedSet	1
	Vector	SynchronizedArrayList	1
Zxing	ArrayList	FastList	3
	HashMap	HashedMap	2

TOMCAT V9 has a substantial number of modifications on **asus**, with a total of 93 recommendations by CT+. For the sake of comparison, TOMCAT V6 had 60 recommendations on **server** and only 13 on **dell**. Being two different versions of TOMCAT, differences in the implementations recommended by CT+ were expected. However, two cases show a contrast in this expectation: `Hashtable` and `HashMap` when comparing **asus** and **server**. `Hashtable` was changed 49 times on **server** (26% of the total recommendations for TOMCAT V6) while it was only changed 8 times on **asus** (8.6% of the total for TOMCAT V9), clearly showing a significant difference. On the other hand, the most common recommendation for both **asus** and **server** was to replace `HashMap` by `HashedMap`. That recommendation was made 47 times on **asus** and 39 for **server**, representing 50% and 65% of all recommendations made for TOMCAT on these devices. Although there are considerable difference between the two TOMCAT versions, it seems like, for several cases, changing `HashMap` for more energy efficient implementation still is an effective recommendation.

Table 9 Recommended collection implementations for the **server** machine and how many times they were recommended.

System	Original	Recommended	# of times
Development Machine: **server**			
Tomcat v6	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

Among all 724 recommendations made to desktop systems, 666 used alternative implementations to JCF, 294 from Apache Commons and 372 from Eclipse Collections. Once again it is possible to observe a trend of replacing well-known collections such as `ArrayList`, `Hashtable`, and `HashMap` by more energy-efficient but less-known alternatives.

Mobile environment Table 10 summarizes the results for the mobile environment. Overall, CT+ made 107 recommendations among the analyzed devices with their effectiveness varying strongly. One of the mobile devices used on our experiments (i.e., **Tab4**) did not present any statistically significant difference between the original and the modified versions. A more thorough discussion about this specific device can be found in Section 8.

The modified versions of PASSWORDGEN on the **S8** and **J7** devices exhibited significant improvements: the modified versions consumed 4.49% and 14.78% less energy than the original ones, with a large effect size. However, **G2** had no recommendations for this specific system (more on this on Section 7).

GOOGLE GSON exhibited a significant improvement of 4.79% 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 modified version consumed 10.16% less energy than the original version on **S8**, the 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 statistically significant way. On the **S8**, albeit statistically significant, the difference was small with the modified version consuming just 0.09% less than the original

Table 10 Results for the mobile environment

System	Improvement	p-value	Mean(J)	Stdev	Effect size
Device: **S8**					
Commons Math	10.16%	1.25^{-8}	92.06	2.59	0.86
			82.70	9.61	
FastSearch	0.09%	1.67^{-3}	35.06	3.32	-0.47
			35.03	1.78	
Google Gson	0.95%	6.42^{-4}	16.45	0.22	0.40
			16.29	0.20	
PasswordGen	4.49%	2.38^{-9}	16.86	0.41	0.90
			16.11	0.65	
Device: **J7**					
Commons Math	-0.33%	2^{-4}	23.82	2.33	-0.56
			23.90	2.62	
Google Gson	4.79%	3.2^{-3}	13.78	1.59	0.44
			13.12	2.67	
PasswordGen	14.78%	6.44^{-9}	12.83	0.90	0.87
			10.94	0.76	
Device: **G2**					
Commons Math	-1.20%	9.0^{-3}	17.22	0.51	-0.41
			17.42	0.14	

Energy results are red for the original versions and green for the modified versions. Energy measured in Joules

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 11 presents the recommendations that CT+ made 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 **dell** 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` with 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.

Table 11 Recommended collection implementations for **S8**, **J7**, and **G2** and how many times they were recommended

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

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.

6.2 Analyzing different profiles

This section describes our experimental environment and results from a study analyzing different energy profiles. Each profile was created simulating a different workload size, in a way to try to analyze how they can affect the energy consumption of Java collections. For this study, only **asus** was used and all the target systems come from the most recent version of the DaCapo benchmark suite, that is, the developer branch of the version 19.07.

The following sections are organized as follows. Section 6.2.1 lists the research questions this study aims to answer; Section 6.2.2 explains the methodology of this study; and Section 6.2.3 presents the results of our experiments.

6.2.1 Research Questions

Following the study on different devices, an open question remained about how changing the energy profile on the same device could impact the recommendations made by CT+. In the previous study, every device had a single energy profile because the time required to create different profiles can be steep. Section 7 discusses the cost of creating energy profiles in more depth. In the follow-up study described in this section, six different profiles were used on the same device. These profiles were built by altering configurations of CT+ during Phase I of the proposed approach (Section 5). We aim to investigate whether the resulting profiles have an impact on the recommendations made by CT+ and the energy efficiency of the target systems after applying these recommendations.

Based on these considerations, with aiming to answer the following research questions (RQs):

RQ3: How much does the workload size impact the energy efficiency of a Java collection implementation?

RQ4: Are the recommendations profile-independent?

6.2.2 Methodology

Through the following experiment, a single device was used, **asus**, described in detail on Table 4. Six target systems present in DaCapo 19.07 were used, that is: BIOJAVA, CASSANDRA, GRAPHCHI, KAFKA, TOMCAT V9 and ZXING. These systems were executed using JDK 8.

To explore the impact of different energy profiles on the energy-efficiency of Java collection implementations, six energy profiles were created for **asus**, namely `N1`, `N2`, `N4`, `N8`, `N16`, and `N32`. Starting with `N1` (the profile used on Section 6.1), created using a specific load size for each API i.e., 15,000 for `Lists`, 18,750 for `Sets`, and 50,000 for `Maps`. We then proceeded to multiplying this load size by a factor of two and then used it to create a new profile, with `N2` having two times the load size of `N1`, `N4` having four times, until the maximum of 32 times the load size of `N1` with `N32`. These profiles were created to simulate different workload sizes a collection may face, with the smaller ones representing lightweight applications that do not depend too much on collections and the bigger ones representing data-structure heavy applications that make more intensive usage of collections.

Load sizes smaller than `N1` made it unreliable to sample the energy consumption for some of the faster operations, such as removing from the tail of a `LinkedList`. The

values of N1 we employed were the lowest loads where it was possible to perform energy measurement, i.e., the results were consistently above zero, with a stable standard deviation, i.e., increasing the load size did not lower it. This phenomenon, in which RAPL is unable to reliably measure energy consumption for small segments of program execution, has been previously reported (Hähnel et al. 2012).

For the experiments, we collected the data from 30 executions of each target system, considering the original and all six modified versions. Each instance was executed in a clean machine state, that is, we restarted the notebook to make sure there were no residual traces from the previous execution. All executions were made in Ubuntu 19.04, booting without a graphic user interface. When experimenting with thread-safe collections, we used four threads for each operation; with non thread-safe collections, only one thread was used.

6.2.3 Study Results

This section first presents the energy consumption results for the original and modified versions. It then proceeds to discuss the recommendations made for each target system when considering the three different profiles. The specific amount of time each system took to finish, the energy consumed, and the source code of the modified applications can be found at https://energycollections.github.io/. In this experiment, considering all six profiles, CT+ made a total of 1711 recommendations.

Figure 2 summarizes the energy consumption of all software systems executed on **asus** in which CT+ made statistically significant recommendations, with the exception of Tomcat v9-HUGE. Out of the six target systems, three were more susceptible to improvements, with high effect size and low p-value across all profiles: BIOJAVA, GRAPHCHI, and ZXING. For two applications, CASSANDRA and KAFKA, CT+ was not able to make any impactful recommendations on any of the six different profiles. For all the scenarios involving these applications, the energy consumption of the original and modified versions did not differ significantly. As a consequence, we do not report the results for CASSANDRA and KAFKA in the remainder of this section. Instead, we focus on statistically significant results. TOMCAT V9 had mixed results, presenting an energy reduction in all profile sizes for the workload LARGE, usually with a small effect size. For the workload size HUGE, TOMCAT V9 presented a reduction in energy consumption of **0.36%** only when using **N32** (with a medium effect).

Among the profiles, there was no overall winner. No profile presented the best results among all different software systems and no profile dominated another one, i.e., for every profile, if a target system had lower energy consumption under profile p_1 than profile p_2, there was some other target system that consumed less under profile p_2. TOMCAT V9, on both workloads sizes, consumed less energy using the recommendations made using N32; GRAPHCHI using N2; ZXING using N1; and BIOJAVA using N8. When looking at the profile sizes, it's possible to have a glimpse of a trend of some systems performing better for smaller profiles (the case of GRAPHCHI) and others performing better for bigger ones (such as BIOJAVA and TOMCAT V9).

Table 12 summarises the data about the recommendations made across all different profiles. The total number of implementation changes differs significantly on each profile,with N4 having the most with 456 recommendations and N16 having the least, with 82. Most of these recommendations changed an implementation from JCF to an alternative collection implementation from Eclipse Collections or Apache Commons Collections. Up to 98% (in the case of N4) of the collections recommended were from alternative sources. On the other hand, N16 had the greater number of recommendations within the JCF, for a total of 49%.

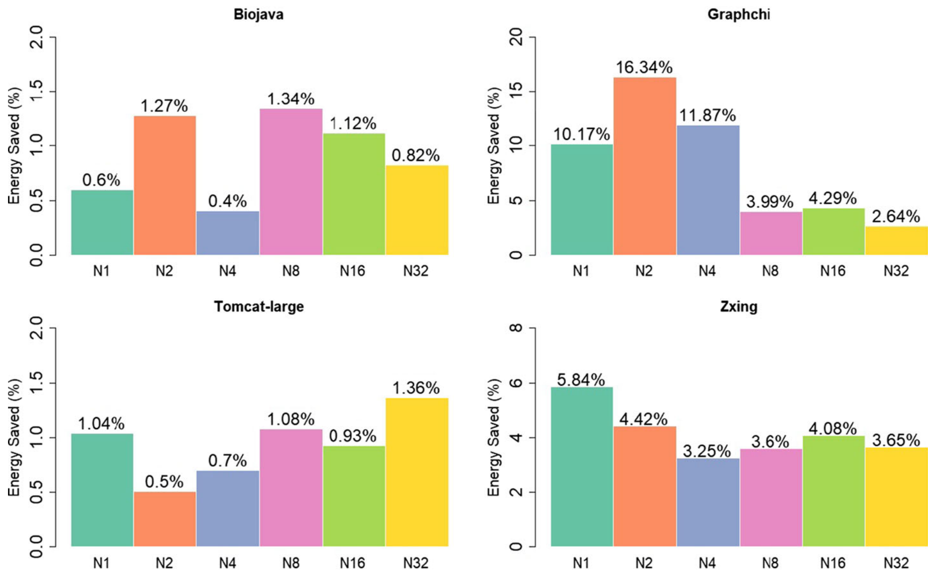


Fig. 2 Percentage of Energy saved by CT+ among the different software systems

Tables 8, 13, 14, 15, 16, and 17 show the specific recommended collections for **asus** among the six different profiles. CT+ often recommended alternatives to `ArrayList` across all devices, for a total of 849 implementations of `ArrayList` being changed by CT+, representing 49.6% of all modifications made across all profiles. That is even clearer for two specific profiles, `N2` and `N4`, having 76.7% and 64.2% respectively of all their modifications being changes from `ArrayList`. On the other hand, `N16` had only two `ArrayList` modifications, representing a total of 2.4% of all modifications made on this profile. Surprisingly, that was not the case for `N32`, which had 210 `ArrayList` implementations being modified, 52.3% of the total modifications of this profile. The second and third most changed `List` implementations across all profile sizes were `CopyOnWriteArrayList` and `LinkedList`, with 47 and 30 changes respectively.

Table 12 Source from the recommendations made to **asus**, sorted by the profile size

Profiles	Total	Sources		
		Java collections	Apache collections	Eclipse collections
`N1`	**247**	26 (10%)	154 (63%)	67 (27%)
`N2`	**352**	19a (5%)	292 (83%)	41 (12%)
`N4`	**456**	7a (2%)	116 (25%)	333 (73%)
`N8`	**173**	38 (22%)	82 (47%)	53 (31%)
`N16`	**82**	40 (49%)	25 (30%)	17 (21%)
`N32`	**360**	41 (11%)	125 (34%)	235 (65%)

The approximated percentage of the total is show between parenthesis

Table 13 Recommended collections for the **asus** on N2

System	Original	Recommended	# of times
Profile: N2			
BioJava	ArrayList	NodeCachingLinkedList	179
	HashSet	UnifiedSet	18
	ArrayList	FastList	6
	HashMap	HashedMap	5
	LinkedList	NodeCachingLinkedList	2
	Hashtable	ConcurrentHashMap	1
Graphchi	ArrayList	NodeCachingLinkedList	4
	HashSet	UnifiedSet	1
Tomcat	ArrayList	NodeCachingLinkedList	64
	HashMap	HashedMap	18
	HashSet	UnifiedSet	10
	Hashtable	ConcurrentHashMap	9
	CopyOnWriteArrayList	SynchronizedArrayList	6
	ConcurrentHashMap	ConcurrentHashMap(EC)	5
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	ConcurrentHashMap	SynchronizedHashMap	1
	LinkedList	NodeCachingLinkedList	1
	TreeSet	TreeSortedSet	1
Zxing	ArrayList	NodeCachingLinkedList	16
	ArrayList	TreeList	1

BIOJAVA showed a special behavior when compared with the other systems. For N4, CT+ recommended 291 changes to BIOJAVA while for N16 the number of recommendations was only 28, less than 10% the recommendations made for N4. Even so, N16 saved more energy than N4 (1.12% and 0.4%, respectively), although in both cases the savings were modest. Not only the recommendations between profiles differ in quantity, but also which collections were recommended by CT+. Among all applications tested on **asus**, on profiles N2 and N4 there was a large number of list implementations that were changed: 283 and 309, respectively. Meanwhile, on profile N16, that number was much smaller: just 15 changes.

When analyzing uses of Set implementations, 35.4% of the recommendations made by CT+ for the N16 profile suggested using some alternative to HashSet. On the other hand, for N1 and N4, only 0.4% and 0.2% of the recommendations involved this collection implementation, respectively. Lastly, HashMap was the Map implementation most often recommended against by CT+. This implementation was frequently changed across all profiles, up to 61.54% on N1, with the solo exception being N2, where its changes represented only 6.53% of the total recommendations. Across all profiles, from all recommendations to change from HashMap, 99.2% were to use HashedMap. Only on 4 cases when analyzing N32, CT+ recommended using UnifiedMap as a replacement for HashMap. This suggests that developers should consider using HashedMap as an alternative to HashMap.

Table 14 Recommended collections for the **asus** on N4

System	Original	Recommended	# of times
Profile: N4			
BioJava	ArrayList	FastList	214
	HashMap	HashedMap	70
	LinkedList	FastList	2
	TreeSet	TreeSortedSet	2
	ArrayList	NodeCachingLinkedList	1
	Hashtable	ConcurrentHashMap	1
	Vector	SynchronizedFastList	1
Graphchi	ArrayList	FastList	7
Tomcat	ArrayList	FastList	69
	HashMap	HashedMap	43
	ConcurrentHashMap	ConcurrentHashMap(EC)	16
	Hashtable	ConcurrentHashMap(EC)	13
	CopyOnWriteArrayList	SynchronizedArrayList	6
	CopyOnWriteArrayList	SynchronizedFastList	2
	LinkedList	TreeList	2
	Vector	SynchronizedFastList	2
	HashSet	UnifiedSet	1
	LinkedList	FastList	1
	TreeSet	TreeSortedSet	1
Zxing	ArrayList	FastList	2

As shown in Section 6.1 and in other papers (Pinto et al. 2014b, 2016), `Hashtable` has poor performance and energy efficiency. Therefore, we expected CT+ to make multiple recommendations of alternative collection implementations aiming to improve uses of `Hashtable`. Surprisingly, though, there were only a few such recommendations. More specifically, CT+ recommended replacing uses of `Hashtable` 8 times for `N1`, 9 times for `N2`, `N8`, `N16`, `N32`, and 13 times to `N4` when analyzing TOMCAT V9 and a single case for each profile for BIOJAVA. An examination of the source code of the target systems reveals that, differently from TOMCAT V9, where there were 49 instances of uses of `Hashtable`, the others either used it scarcely (2 cases for BIOJAVA, CASSANDRA, and KAFKA) or did not use it at all (GRAPHCHI and ZXING). This decreased `Hashtable` usage in more modern software corroborates the results from our queries on GitHub projects using Java collections, presented in Table 1.

7 Discussion

This section discusses in more depth the results from the two studies presented in Sections 6.1.3 and 6.2.3. This section is organized as follows. We start by examining the

Table 15 Recommended collections for the **asus** on N8

System	Original	Recommended	# of times
Profile: N8			
Biojava	HashMap	HashedMap	45
	HashSet	LinkedHashSet	18
	ArrayList	FastList	15
	LinkedList	ArrayList	2
	ArrayList	LinkedList	1
	HashTable	ConcurrentHashMap(EC)	1
Graphchi	ArrayList	FastList	2
	HashSet	LinkedHashSet	1
Tomcat	HashMap	HashedMap	34
	ConcurrentHashMap	ConcurrentHashMap(EC)	19
	HashSet	LinkedHashSet	9
	Hashtable	ConcurrentHashMap(EC)	9
	CopyOnWriteArrayList	SynchronizedArrayList	4
	CopyOnWriteArrayList	SynchronizedFastList	2
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	ArrayList	FastList	1
	HashSet	UnifiedSet	1
	LinkedList	ArrayList	1
Zxing	ArrayList	FastList	3
	ArrayList	TreeList	1

implementations of JCF, Apache Collections, and Eclipse Collections and how they interact during our experiments. We then proceed by analyzing the energy consumption of the most popular implementations, answering **RQ1**, and discussing the importance of analyzing different devices when running experiments on energy efficiency, answering **RQ2**. We also discuss how the optimization of the collection implementations change based on the expected workload, answering **RQ3** and the impact that different energy profiles have on the expected consumption on an optimized application, answering **RQ4**. Finally, we illustrate how some implementations dominate others and explain the employed procedure to optimize the creation of energy profiles.

JCF recommendations The majority of the CT+ recommendations were for collection implementations not in the JCF. Considering only the statistically significant occurrences, out of 724 recommendations made in the desktop environment in the first study, only 724JCF suggested the use of JCF collections (8% of the recommendations). When analyzing the data on the study about different energy profiles, out of 1711 recommendations, 171 were originated from JCF (10% of the recommendations). This means that overall, in the desktop environment, across all 724CrossStudy recommendations made, only 9.2% of

Table 16 Recommended collections for the **asus** on `N16`

System	Original	Recommended	# of times
Profile: `N16`			
Biojava	HashSet	LinkedHashSet	18
	HashMap	HashedMap	5
	LinkedList	ArrayList	2
	ArrayList	NodeCachingLinkedList	1
	Hashtable	ConcurrentHashMap(EC)	1
	TreeSet	TreeSortedSet	1
Graphchi	HashSet	LinkedHashSet	1
Tomcat	HashMap	HashedMap	16
	HashSet	LinkedHashSet	10
	Hashtable	ConcurrentHashMap(EC)	9
	ConcurrentHashMap	ConcurrentHashMap(EC)	6
	CopyOnWriteArrayList	SynchronizedArrayList	6
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	LinkedList	ArrayList	1
Zxing	ArrayList	TreeList	1

the CT+ recommendations suggested the use of JCF implementations. 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). However it is worth noting that none of the cases where energy was saved on **J7** used JCF implementations. If we aggregate over all of these recommendations, the JCF was recommended in just 10.4% of the cases.

Popular collections and energy efficiency Our results indicate that there seem to be more energy-efficient alternatives to some popular collection implementations. In the desktop environment, CT+ recommended replacing 184 uses of `Hashtable`, 654 uses of `HashMap`, 138 uses of `HashSet`, 38 uses of `LinkedList`, and 1027 uses of `ArrayList`. Overall, those recommendations amount to 93.2% of all the recommendations in cases where there was a statistically significant difference in energy consumption. This percentage is consistent with the popularity of those JCF collections (Table 1); since they are used often, there will be many recommendations to replace them with alternatives. Some of these commonly used implementations were not recommended by CT+, e.g., `HashMap`, and `HashSet`, while others were very rarely recommended, such as `LinkedList` (3 times), `Hashtable` (3 times), and `ArrayList` (12 times). `Hashtable`, in particular, replaced `ConcurrentHashMap` on `N1`. For as much performance and scalability problems `Hashtable` may have, it seems like it can still perform better than other implementations for a very small number of elements.

Out of the 12 times `ArrayList` was recommended, 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

Table 17 Recommended collections for the **asus** on `N32`

System	Original	Recommended	# of times
Profile: `N32`			
Biojava	ArrayList	FastList	145
	HashMap	HashedMap	80
	HashSet	LinkedHashSet	18
	HashMap	UnifiedMap	3
	LinkedList	FastList	2
	ArrayList	LinkedList	1
	Hashtable	ConcurrentHashMap	1
Graphchi	ArrayList	FastList	2
	HashSet	LinkedHashSet	1
	HashMap	HashedMap	1
Tomcat	ArrayList	FastList	48
	HashMap	HashedMap	40
	ConcurrentHashMap	ConcurrentHashMap(EC)	14
	HashSet	LinkedHashSet	10
	Hashtable	ConcurrentHashMap(EC)	7
	CopyOnWriteArrayList	SynchronizedLinkedList	3
	CopyOnWriteArrayList	SynchronizedArrayList	2
	CopyOnWriteArrayList	Vector	2
	Hashtable	ConcurrentHashMap	2
	LinkedList	TreeList	2
	HashMap	UnifiedMap	1
	LinkedList	ArrayList	1
Zxing	ArrayList	FastList	13
	ArrayList	NodeCachingLinkedList	1
	HashMap	HashedMap	1

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 of modifying popular collections 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.

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 List.insert(value) and List.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.*"[13]. 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, CT+ recommended ten ArrayList instances to be changed to FastList and one to NodeCachingLinkedList when analyzing Xalan on **dell**. However, for the same system on **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 mobile devices, even if the modified versions consumed less energy. On the other hand, on **dell**, 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 **dell**, 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 **dell**, 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.

For this topic, results from **asus** were not analyzed because they are not comparable. The only target system used on it and on the other two desktop devices, namely **dell** and **server**, was Tomcat, but with a different version. Besides that, we used six different profiles on **asus**, producing different recommendations, and making the comparison infeasible. The next sections will give more details about our findings on the different profiles for **asus**.

The best implementation is workload dependent With the experiments on **asus**, we used six different profiles to try to simulate the different scenarios that the application could be submitted to. The amount of energy saved on different applications was heavily influenced by the profile being used, specially in two cases: GraphChi, with 12.73% of energy was saved using N2 but only 2.64% using N32; and BioJava, with 1.34% of energy was saved

[13] Available at: https://www.eclipse.org/collections/

using N8 but only 0.40% using N4. These results indicate that, even though profile creation is an application-independent step of the proposed approach, knowledge about actual usage profiles can be leveraged to produce more useful profiles.

Recommendations applied to TOMCAT V9 using the six different profiles on the LARGE workload resulted in a positive impact on energy efficiency with statistical significance. On the other hand, for the HUGE workload, the results did not have statistical significance for five profiles sizes (N1, N2, N4, N8, and N16), having only a positive impact when using the biggest profile size, N32. Comparing the recommendations made to TOMCAT V9-HUGE using N32 (Table 17) and using the smaller sized profiles, such as N1 (Table 8) and N2 (Table 13), we can see that they differ greatly.

Although both LARGE and HUGE were bigger than normal workload sizes, the difference between these two was enough to make CT+ unable to recommend better collections implementations to any of the profile sizes with the only exception being N32.

Energy profiles also matter As the profiles were created to represent different scenarios, a different behavior was expected. In this topic, we take a deeper look at the implementations recommended for these different profiles on **asus**, in particular as replacements to uses of ArrayList and HashSet.

Across all profiles, 90.8% of list modifications were changes from ArrayList to another implementation. In particular, FastList was the implementation of choice by CT+ in 68% of the cases (577 cases out of 849). On the other hand, out of 270 changes on N2 from ArrayList, only 6 were to FastList. On this particular profile size, the most often used implementation to replace ArrayList was NodeCachingLinkedList, with a total of with 263 changes. The main reason behind the substantial difference in recommendations between the different profiles is the distinct operations used by each application. Taking a deeper look at the behavior of those two collections, we noticed that on N2, ArrayList has higher energy consumption in five out of ten operations than FastList and NodeCachingLinkedList. When comparing between themselves, FastList and NodeCachingLinkedList had an even number of operations where they performed better, five each. Even being the best replacement for ArrayList on every other profile size, for N2, NodeCachingLinkedList exhibited better results than FastList.

The modifications from ArrayList on N2 were mostly focused on one application: BIOJAVA. On this system, two operations were heavily used and had a greater impact on CT+ recommendations: List.insert(value) and List.iteration(iterator). NodeCachingLinkedList (on profile N2) consumed less energy than ArrayList for these two operations. In contrast, when looking at profile N8 and N16, ArrayList did not have a single implementation that had lower consumption for these two operations. These two are precisely the profiles with the smaller number of modifications from ArrayList: 23 changes on N8 and only 2 on N16.

HashSet had a total of 118 modifications across the different profiles on **asus**. Most of these changes replaced it by one of two particular implementations: LinkedHashSet (72%) and UnifiedSet (26%). These changes were not equally divided among the profiles. For smaller profiles, CT+ recommended UnifiedSet to replace HashSet on every change on N1, N2 and N4. For bigger profiles, CT+ recommended LinkedHashSet to replace HashSet on every change on N16 and N32, and all but one on N8 (the only exception being a recommendation to use UnifiedSet).

A change in the expected scenario (represented here by a change in profile sizes) had a sizable impact on the recommended implementations and in the energy efficiency results.

As researchers, we could only faintly foresee what kind of scenario would better represent a normal usage pattern for a specific application. On the other hand, developers would have an easier time figuring it out how much work the most important parts of the application are expected to have. This kind of information could be very valuable for the creation of the energy profile and could improve even more the efficiency of CT+, allowing us to combine different profiles to make the recommendations.

Dominance among collection implementations Out of the 39 possible implementations available to CT+ only 20 were recommended. When trying to understand this behavior, we observed that some collection implementations consistently dominate (Peterson 2009) 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 $(T, env, e_1, e_2, ..., e_n)$ and $(T, env, e'_1, e'_2, ..., e'_n)$, 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 implementation, it will never be recommended by CT+.

Figure 3 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`, `ConcurrentHashMap(EC)`, and the `Synchronized` versions of `LinkedHashMap` and `UnifiedMap`. These are the collections that are not dominated by any other collection. Furthermore, as the figure shows, `Hashtable` is dominated by `ConcurrentHashMap(EC)`, even though `Hashtable` itself also dominates `Synchronized TreeMap`. Therefore, in **server**, instances of `Synchronized TreeMap` and `Hashtable` are never recommended, in favor of `ConcurrentHashMap(EC)`. More specifically, we observed that `Hashtable` was dominated on **dell**, **server**, and on every mobile device that we experimented with. This result, combined with the well-known scalability limitations of this collection (Pinto et al. 2016), 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 devices.

Among all implementations, `ConcurrentHashMap` shows a particular behavior that is worth mentioning. That implementation was replaced by more efficient alternatives 79 times on the desktop environment, 70 out of 79 cases for the Eclipse Collections version (i.e., `ConcurrentHashMap(EC)`). Even so, `ConcurrentHashMap`

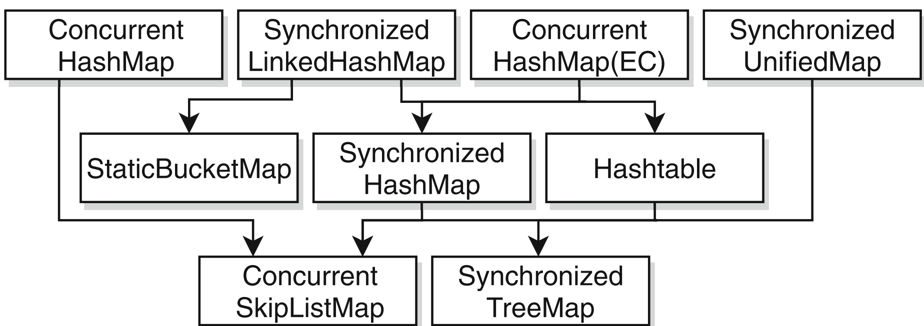


Fig. 3 Order of dominance between the thread-safe Map implementations on **server**. Arrows point from the dominating collection to the dominated one

was also recommended 37 times, every single time replacing `Hashtable`. This illustrates that even if some implementation is usually recommended over another one (e.g., `ConcurrentHashMap(EC)` recommendations over `ConcurrentHashMap`), as long as this implementation is not dominated, there will be cases where the generally worse implementation may still perform better.

Scaling up profile creation We used two different profilers in this work, one for mobile devices and one for desktop devices, as described in Section 5. During our experiments, we noticed that some factors could make it unfeasible to create profiles at a larger scale. If left unchecked, the original process of creating the energy profiles can take a long time, i.e., hours for desktop devices and days for mobile devices. This is due to the enormous variation in the execution time of operations for different collections.

On the one hand, some operations are so fast that it is necessary to increase the number of times they are executed during profiling in order to obtain reliable energy measurements, e.g., insertions at the beginning of a `LinkedList`. On the other hand, some operations are so slow that we need to reduce the number of executions, e.g., updates to `CopyOnWriteArrayList` when the list has many elements. For example, when creating the profile `N16`, the operation `insert(start)` on `LinkedList` would be 554 times faster than on `ArrayList` while `insert(value)` would be 934 times slower on `CopyOnWriteArrayList` than on `Vector`. To address this issue, we employed two strategies, described below.

- *Excluding collections that were dominated.* First we executed each operation for each implementation three times, measured and collected the energy consumption of those operations. In the case where we found one collection implementation exhibiting domination over another, the dominated collection was not included as an option for recommendation. As an example, when creating `N2` on **asus**, there were 13 initial possibilities for Sets. Out of that initial pool, four implementations, that is, `LinkedHashSet`, and the `Synchronized` versions of `UnifiedSet`, `TreeSet`, and `LinkedHashSet`, were excluded because they were dominated by other implementations. In this case, these techniques represent savings of at least 30% of the time to generate the `Set` portion of the profile `N2`, potentially more. Because the dominated implementations would never be recommended by CT+, they can be safely removed from our recommendation pool without reducing CT+ capacity of improving the energy efficiency of the application.
- *Using timeouts.* Because of the long time it took to execute some operations on mobile devices, e.g., insertions on instances of `CopyOnWriteArrayList`, very expensive operations were discarded based on the time it took for them to complete. To define which operations should not be measured, a single warmup session was executed, collecting the energy consumption for each operation on each collection implementation. Each operation implementation was executed in sequence and the values for the fastest ones were stored, separated by thread-safety and API (e.g., `insert(start)` for thread-safe `Lists`). Because some of those operations could take a long time to finish, we established a threshold on the number of times that an operation could be slower than the fastest ine. Any operation slower than that was stopped in the middle of its execution.[14]

[14]On our experiments, the threshold was 100 times slower

We assumed that such a large difference is unlikely to stem from random performance fluctuations. Since CT+ needs estimated energy consumption measurements per operation to recommend the implementations, we used the energy consumed by the fastest operation multiplied by our threshold. This approach loses some energy consumption information during profile construction. On the one hand, this means that sub-optimal collection implementations may end up being recommended by CT+ because we end up underestimating the cost of the very expensive operations. On the other hand, in our experiments, we have observed that scenarios where such collection implementations would thrive, e.g., `CopyOnWriteArrayList` in a scenario where a large collection is subject to many concurrent accesses almost exclusively for reading, were rare. As a consequence, collections in which most of the operations were expensive almost never got recommended.

8 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. 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 433k lines of code and has been used in multiple studies (Pinto et al. 2016; Hasan et al. 2016; Pereira et al. 2018).

Even though we observed an overall good energy savings with our tool, for some software systems it was not possible to reduce the energy consumption reported in other studies. We hypothesize this happens due to the care we took of preventing thread-safety problems due to recommendations that ignore this aspect. We checked that among the recommendations in the study of Pereira et al. (2018) there were cases where a thread-safe collection was replaced by a non-thread-safe one. Similarly, our tool does not guarantee thread-safety when thread-safe collections are performing compounded operations (Lin and Dig 2015) (e.g., verifying if an item is stored in a collection before adding it). In other words, it does not break thread-safety requirements, but, at the same time, it cannot guarantee thread-safety for non-thread-safe operations. The software construct versions (such as libraries and applications) may influence the recommendations made by CT+.

The Java Development Kit version may have a non-negligible influence on our results. Through this article, JDK 6 was used for the study presented in Section 6.1 and JDK 8 for Section 6.2. Other versions of Java may have different implementations of the collections used on this article and this may lead to different results. The configurations and source code are available at https://energycollections.github.io/.

Another limitation of this work is the way loops are accounted for in Phase II of the proposed approach (Section 5). Estimating loop counts is difficult in a high-level programming language such as Java, where collections are allocated dynamically, usually based on information from outside the system's source code (Rodrigues et al. 2014). On the one hand, the approach employed in this work is very cheap and takes loop nesting into account. Nevertheless, it is inherently imprecise; in extreme cases, it may consider that loops that are executed millions of times have the same impact on energy consumption as loops that execute just a dozen times. Even though these two situations would be equivalent from an asymptotic perspective, they differ significantly in practice. Exploring different approaches to account

for loops, recursion, and API functions that encapsulate repetition, e.g., map/reduce, is left for future work.

The methodology used to collect energy consumption may have an impact on the data presented in this study. The energy measurement was made at the application level for the mobile devices and at the system level for the desktop devices. As explained in Sections 6.1.2 and 6.2.2, we mitigated the influence of external factors by executing only the application under analysis on mobile devices and executing our experiments on an operating system without a graphic user interface. Nevertheless, the energy data presented in this paper may differ from other devices or operating systems.

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.

8.1 Results without statistical significance.

For some applications, applying the recommendations made by CT+ to a target system did not yield a more energy efficient, at least not enough to exhibit a statistically significant difference to the original version. That was the case for two systems on Section 6.1 desktop environment (XISEMELE and TWFBPLAYER) and also two on Section 6.2 (CASSANDRA and KAFKA). On the mobile environment there were four in this situation: GOOGLE GSON and PASSWORDGEN on **G5**; FASTSEARCH on every device except **S8**; and XSTREAM on every device. The only device where there was no statistically significant difference in the energy consumption of the original and modified versions of the target systems was **Tab4**.

Table 18 shows the results of the experiments on **Tab4**. Albeit 194 recommendations were made on five different software systems for **Tab4**, none of them had statistically significant results. Unlike other devices from our mobile pool, **Tab4** is a special device, being a tablet and not a smartphone. The idea was to try to investigate a distinct type of device and see if the applications running on it could also be optimized by CT+. Even though Table 18 suggests that the CT+ recommendations yielded positive results for most of the apps, the effect sizes were all small or negligible and there was no statistical difference.

We hypothesize that the reason for this result was the very high standard deviation present in the collected samples. Even though the standard deviations we observed for mobile devices (Table 10) was in general much higher than for the desktop devices (Table 6), they were even higher for **Tab4**. Considering the two versions of each of the five apps we have analyzed on **Tab4**, only one version exhibited a standard deviation lower than 10% of the mean energy consumption (the original GOOGLE GSON) and the worst-case scenario reached more than 30% (the original PASSWORDGEN), as shown in Table 18.

When investigating the reasons behind this result, we noticed that the battery was discharging at an inconsistent rate, even when in an idle state. We hypothesize that this inconsistency stems from the device's age. For reference, Table 4 shows the age of the devices when executed the experiments on them. On older devices, the energy consumption difference between the original and the modified version seems to be more indistinguishable, that is, the impact made by optimizing the collections were not represented in the final energy consumption. Although these experiments involved only 4 mobile devices, they suggest that battery age can impact energy measurements. Since real-world device usage involves both old and new devices, simply using newer devices is not an appropriate solution. Instead, we recommend that future studies include the **age of the devices (or their batteries)** when reporting experimental results.

Table 18 Results for **Tab4**

System	Improvement	p-value	Mean	Stdev	Effect size
Device: Tab4					
Commons Math	1.99%	0.39	23.62	3.31	0.10
			23.04	4.47	
Google Gson	1.58%	0.80	56.01	5.04	0.05
			55.15	6.77	
Xstream	6.16%	0.24	26.93	6.48	0.17
			25.20	6.91	
PasswordGen	6.23%	0.24	28.80	9.65	0.03
			27.36	6.77	
FastSearch	4.44%	0.80	50.11	7.34	0.27
			47.08	7.49	

Energy results are red for the original versions and green for the modified versions. Energy measured in Joules

9 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 implementations. We evaluated two different experiments studies, analyzing the influence of devices and energy profiles on software systems' energy-efficiency using Java collections. Overall, we executed our tool in seven different devices running seventeen different software systems (two mobile, twelve desktop, and three on both environments), and six other energy profiles for a total of 64 software versions.

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 16.34%. Overall, CT+ made a total of 2295 recommendations that lead to a statistically significant impact on the energy-efficiency.

Our results suggest that some of the most popular collections implementations (e.g., `ArrayList`, `HashMap`, `HashSet`, and `Hashtable`) are often not the most energy-efficient ones; that implementations have different energy behavior while dealing with different quantity of data and developers should try to choose a collection based on the expected work; and that the energy-efficiency of the collections changed based on the devices they were running. As *future work*, we plan to submit patches to the projects applying the modifications made by CT+. We also 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 paper acknowledges the support of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2020-1-PT01-KA203-078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education". The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein. This research was partially funded by CAPES/Brazil (88887.364121/2019-00), CNPq/Brazil (304755/2014-1, 406308/2016-0, 465614/2014-0, 309032/2019-9), FACEPE/Brazil (APQ-0839-1.03/14, 0388-1.03/14, 0592-1.03/15).

References

Aggarwal K, Zhang C, Campbell JC, Hindle A, Stroulia E (2014) The power of system call traces: predicting the software energy consumption impact of changes. In: Proceedings of 24th Annual International Conference on Computer Science and Software Engineering, CASCON 2014. IBM / ACM, pp 219–233

Andrae A (2017) Total consumer power consumption forecast. Nordic Digital Business Summit 10

Baldwin CY, Clark KB Design rules, volume 1: The power of modularity, vol 1, 1st edn. The MIT Press

Barrett E, Bolz-Tereick CF, Killick R, Mount S, Tratt L (2017a) Virtual machine warmup blows hot and cold. Proc ACM Program Lang 1(OOPSLA):52:1–52:27. https://doi.org/10.1145/3133876

Bennett J, Lanning S, Netflix N (2007b) The netflix prize. In: In KDD Cup and Workshop in conjunction with KDD

Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B (2006) 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. ACM, New York, pp 169–190. https://doi.org/10.1145/1167473.1167488

Chen Q, Grosso P, Van Der Veldt K, De Laat C, Hofman R, Bal H (2011) Profiling energy consumption of VMs for green cloud computing. In: Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011, pp 768–775

Chowdhury SA, Hindle A (2016) Characterizing energy-aware software projects: Are they different? In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16. ACM, New York, pp 508–511. https://doi.org/10.1145/2901739.2903494

Chowdhury SA, Hindle A, Kazman R, Shuto T, Matsui K, Kamei Y (2019a) Greenbundle: An empirical study on the energy impact of bundled processing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp 1107–1118

Chowdhury S, Borle S, Romansky S, Hindle A (2019b) Greenscaler: Training software energy models with automatic test generation. Empir Softw Engg 24(4):1649–1692. https://doi.org/10.1007/s10664-018-9640-7

Cliff N (1993) Answering ordinal questions with ordinal data using ordinal statistics. Multivar Behav Res 31:331–350. https://doi.org/10.1207/s15327906mbr3103_4

Costa D, Andrzejak A, Seboek J, Lo D (2017) Empirical study of usage and performance of java collections. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17. ACM, New York, pp 389–400. https://doi.org/10.1145/3030207.3030221

Cruz L, Abreu R (2017) Performance-based guidelines for energy efficient mobile applications. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft '17. IEEE Press, pp 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

Duarte LM, da Silva Alves D, Toresan BR, Maia PH, Silva D (2019) A model-based framework for the analysis of software energy consumption. In: Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019. Association for Computing Machinery, New York, pp 67–72. https://doi.org/10.1145/3350768.3353813

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. SIGPLAN Not 42(10):57–76. https://doi.org/10.1145/1297105.1297033

Georgiou S, Spinellis D (2020) Energy-delay investigation of remote inter-process communication technologies. J Syst Softw 162:110506. https://doi.org/10.1016/j.jss.2019.110506, http://www.sciencedirect.com/science/article/pii/S0164121219302808

Hähnel M, Döbel B, Völp M, Härtig H (2012) Measuring energy consumption for short code paths using rapl. SIGMETRICS Perform Eval Rev 40(3):13–17. https://doi.org/10.1145/2425248.2425252

Hao S, Li D, Halfond WGJ, Govindan R (2013) Estimating mobile application energy consumption using program analysis. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, pp 92–101. http://dl.acm.org/citation.cfm?id=2486788.2486801

Hasan S, King Z, Hafiz M, Sayagh M, Adams B, Hindle A (2016) Energy profiles of java collections classes. In: Proceedings of the 38th International Conference on Software Engineering, New York, pp 225–236. https://doi.org/10.1145/2884781.2884869

Lee J, Chon Y, Cha H (2015) Evaluating battery aging on mobile devices. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp 1–6

Li D, Hao S, Gui J, Halfond WGJ (2014a) An empirical study of the energy consumption of android applications. In: 30th IEEE International Conference on Software Maintenance and Evolution, pp 121–130

Li D, Tran AH, Halfond WGJ (2014b) Making web applications more energy efficient for OLED smartphones. In: 36th International Conference on Software Engineering (ICSE'2014). ACM, pp 527–538

Lima LG, Soares-Neto F, Lieuthier P, Castor F, Melfe G, Fernandes JP (2016) 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), vol 1, pp 517–528

Lin Y, Dig D (2015) A study and toolkit of CHECK-THEN-ACT idioms of java concurrent collections. Softw Test Verif Reliab 25(4):397–425

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Oliveto R, Di Penta M, Poshyvanyk D (2014) 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. ACM, New York, pp 2–11. https://doi.org/10.1145/2597073.2597085

Linares-Vásquez M, Bavota G, Cárdenas CEB, Oliveto R, Di Penta M, Poshyvanyk D (2015) 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. ACM, New York, pp 143–154

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Penta MD, Oliveto R, Poshyvanyk D (2018) Multi-objective optimization of energy consumption of guis in android apps. ACM Trans Softw Eng Methodol 27(3):14:1–14:47

Liu K, Pinto G, Liu D (2015) Data-oriented characterization of application-level energy optimization. In: Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering, FASE'15

Lyu Y, Gui J, Wan M, Halfond WGJ (2017) An Empirical Study of Local Database Usage in Android Applications. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)

Manotas I, Pollock L, Clause J (2014) Seeds: A software engineer's energy-optimization decision support framework. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp 503–514. https://doi.org/10.1145/2568225.2568297

Manotas I, Bird C, Zhang R, Shepherd D, Jaspan C, Sadowski C, Pollock L, Clause J (2016) An empirical study of practitioners' perspectives on green software engineering. In: ICSE, pp 237–248

Mcintosh A, Hassan S, Hindle A (2019) What can android mobile app developers do about the energy consumption of machine learning? Empir Softw Engg 24(2):562–601. https://doi.org/10.1007/s10664-018-9629-2

Mingay S (2007) Green IT: The new industry shockwave. Gartner RAS Core Research Note G00153703

Oliveira W, Oliveira R, Castor F (2017) A Study on the Energy Consumption of Android App Development Approaches. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Oliveira W, Oliveira R, Castor F, Fernandes B, Pinto G (2019) Recommending energy-efficient java collections. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp 160–170

Pang C, Hindle A, Adams B, Hassan AE (2016) What do programmers know about software energy consumption? IEEE Softw 33(3):83–89. https://doi.org/10.1109/MS.2015.83

Pereira R, Couto M, Saraiva J, Cunha J, Fernandes JP (2016) 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. ACM, New York, pp 15–21. https://doi.org/10.1145/2896967.2896968

Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, Saraiva J (2017) Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017. ACM, New York, pp 256–267. https://doi.org/10.1145/3136014.3136031

Pereira R, Simão P, Cunha J, Saraiva J (2018) jStanley: Placing a Green Thumb on Java Collections. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018. ACM, New York, pp 856–859. https://doi.org/10.1145/3238147.3240473

Peterson M (2009) Decisions under ignorance. In: An Introduction to Decision Theory, Cambridge Introductions to Philosophy. Cambridge University Press, pp 40–63

Pinto G, Castor F, Liu Y (2014a) Mining questions about software energy consumption. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 22–31

Pinto G, Castor F, Liu YD (2014b) Understanding energy behaviors of thread management constructs. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14, pp 345–360

Pinto G, Liu K, Castor F, Liu YD (2016) A comprehensive study on the energy efficiency of java thread-safe collections. In: ICSME

Rocha G, Castor F, Pinto G (2019) Comprehending energy behaviors of java i/o apis. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–12

Rodrigues RE, Alves P, Pereira F, Gonnord L (2014) Real-world loops are easy to predict: a case study. In: Workshop on Software Termination (WST'14)

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) 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

Romansky S, Borle NC, Chowdhury S, Hindle A, Greiner R (2017) Deep green: Modelling time-series of software energy consumption. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 273–283

Saborido R, Morales R, Khomh F, Guéhéneuc Y-G, Antoniol G (2018) Getting the most from map data structures in Android. Empir Softw Eng. https://doi.org/10.1007/s10664-018-9607-8

Sahin C, Cayci F, Gutiérrez ILM, Clause J, Kiamilev F, Pollock L, Winbladh K (2012) Initial explorations on design pattern energy usage. In: 2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings, pp 55–61

Sahin C, Pollock L, Clause J (2014) How do code refactorings affect energy usage? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, pp 36:1–36:10

Shapiro SS, Wilf MB (1965) An Analysis of Variance Test for Normality (complete samples). Biometrika 52(3-4):591–611

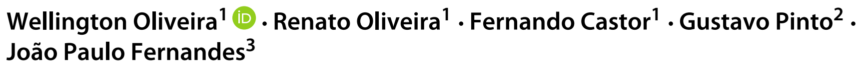
Šimunić T, Benini L, De Micheli G, Hans M (2000) Source code optimization and profiling of energy consumption in embedded systems. In: Proceedings of the International Symposium on System Synthesis, pp 193–198

Wan M, Jin Y, Li D, Gui J, Mahajan S, Halfond WilliamGJ (2017) Detecting display energy hotspots in android apps. Softw Test Verif Reliab 27(6):16–35

Wilks DS (2011) Statistical methods in the atmospheric sciences. Elsevier Academic Press, Amsterdam. https://www.amazon.com/Statistical-Atmospheric-Sciences-International-Geophysics/dp/0123850223/ref=pd_bxgy_14_img_3?_encoding=UTF8&psc=1&refRID=ESPQQ0R2PB1TP1VJSGCZ

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations



Wellington Oliveira[1] · Renato Oliveira[1] · Fernando Castor[1] · Gustavo Pinto[2] · João Paulo Fernandes[3]

Renato Oliveira
ros3@cin.ufpe.br

Fernando Castor
castor@cin.ufpe.br

Gustavo Pinto
gpinto@ufpa.br

João Paulo Fernandes
jpaulo@fe.up.pt

1 Informatics Center, Federal University of Pernambuco (UFPE), Pernambuco, Brazil

2 Faculty of Computing of the Federal University of Pará (UFPA), Pernambuco, Brazil

3 CISUC, Faculty of Engineering of the University of Porto (FEUP), Porto, Portugal