Pós-Graduação em Ciência da Computação

Renato Oliveira dos Santos

LEVERAGING COLLECTION DIVERSITY TO IMPROVE ENERGY EFFICIENCY

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2019

Renato Oliveira dos Santos

LEVERAGING COLLECTION DIVERSITY TO IMPROVE ENERGY EFFICIENCY

A M.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering
Advisor: Fernando José Castor de Lima Filho

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S237l Santos, Renato Oliveira dos
Leveraging Collection Diversity to Improve Energy Efficiency/ Renato Oliveira dos Santos – 2019.
67 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndice.

1. Engenharia de Software. 2. CECOTool. 3. Energy profiling. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-093

Renato Oliveira dos Santos

"Leveraging Collection Diversity to Improve Energy Efficiency""

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 15/02/2019.

BANCA EXAMINADORA

____

Prof. Dr. Kiev Santos da Gama
Centro de Informática / UFPE

____

Prof. Dr. Lincoln Souza Rocha
Departamento de Computação / UFC

____

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE
(**Orientador)**

I dedicate this thesis to all my family, friends and professors who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

I wish to thank my advisor Fernando José Castor de Lima Filho for his assistance on my journey during this research, from its beginning to its completion, for being a partner, and for understading any personal matters that unfortunatelly slowed things down.

I wish to thank the postgraduate student Wellington de Oliveira Junior for helping me with my research and for allowing me to cowork with him.

I wish to thank Gustavo Henrique Lima Pinto and José Benito Fernandes de Araújo Neto for allowing us to extend their work and for helping us to ramp up with what has already been done.

I wish to thank my family who supported me during this journey and for never letting me turn back.

ABSTRACT

The increase in the use of pocket devices, such as smartphones and tablets, and the growth of embedded systems and data centers have moved the scientific community towards research lines involving the area of energy consumption. Many of these studies were initially focused on hardware, such as CPU and memory, and on operational systems, as means of improving the energy efficiency. This research, instead of focusing on these infrastructure components, proposes solutions to reduce the energy consumption of the applications that run on this infrastructure. In particular, this work proposes a tool called CT+, which statically analyses software systems that are written in Java and that use collections intensively, and proposes alternative collections implementations that are more efficient regarding energy consumption. The tool is an extension of another tool proposed in a previous work, called CECOTool, but it implements a series of improvements that solve limitations of the original approach. More specifically, depending on the context of use, it is capable of (i) recommending either collections that are safe for multiple threads or collections that are not (but that tend to be more efficient), (ii) do recommendations taking into account two other commonly used collections libraries, the Eclipse Collections and the Apache Commons Collections, (iii) distinguish operations executed on the beginning, middle or ending of a sequential structure; (iv) automatically apply the recommendations. Furthermore, CT+ makes use of points-to analysis to identify objects that are passed as parameters to other methods, making it possible for the recommendations to take into account the use of the same collection in different methods. In addition, besides being able to recommend to desktop or server applications, CT+ is also capable of recommending to mobile applications that target the Android platform. The CT+ evaluation shows how it was possible to improve the results of the original study by doing a comparison of the energy reduction on the two originally used benchmarks, reducing 5.49% of energy consumption against 3.49% on Xalan application and 4.83% against 4.37% on Tomcat. The effectiveness of CT+ in recommending collections for Android application was evaluated on the context of three different devices. It was possible to reach a reduction of energy consumption of up to 14.73%.

Keywords: CECOTool. CT+. Java Collections Framework. Energy profiling.

RESUMO

A maior utilização de dispositivos de bolso, como *smartphones* e *tablets*, e o crescimento de sistemas embarcados e de *data centers*, têm levado a comunidade científica a iniciar linhas pesquisa na área de consumo de energia. Muitos desses estudos inicialmente tiveram foco em *hardware*, como *CPU* e memória *RAM*, e em sistemas operacionais, como forma de melhorar a eficiência energética. Este trabalho, ao invés de focar nestes componentes de infraestrutura, visa propor soluções para reduzir o consumo de energia das aplicações que rodam sobre essa infraestrutura. Em particular, propõe uma ferramenta chamada CT+, que analisa estaticamente sistemas de software escritos na linguagem Java e que usem coleções intensamente e propõe implementações alternativas de coleções que sejam mais eficientes do ponto de vista energético. A ferramenta é uma extensão de uma ferramenta proposta em um trabalho anterior, chamada CECOTool, mas implementa diversas melhorias que atacam limitações da abordagem original. Mais especificamente, dependendo do contexto de uso, ela é capaz de (i) recomendar tanto coleções seguras para múltiplas threads quanto coleções que não são seguras (mas que tendem a ser mais eficientes), (ii) fazer recomendações levando em conta mais duas bibliotecas de coleções muito usadas na prática, a Eclipse Collections e a Apache Commons Collections, (iii) distinguir operações realizadas no começo, no meio e no final de uma estrutura sequencial; (iv) aplicar automaticamente as recomendações realizadas. Além disso, CT+ utiliza-se de análise points-to para identificar objetos passados como parâmetros de métodos, o que torna possível que recomendações levem em conta usos de uma mesma coleção em diferentes métodos. Complementarmente, CT+ é capaz de realizar recomendações tanto para aplicações que rodam em máquinas desktop ou servidores quanto para aplicações móveis que tenham como alvo a plataforma Android. A avaliação de CT+ mostra como foi possível superar os resultados do estudo original fazendo um comparativo da melhoria de consumo de energia obtida nos dois benchmarks originalmente utilizados, economizando 5.49% de energia contra 3.49% na aplicação Xalan e 4.83% contra 4.37% no Tomcat. A eficácia de CT+ para recomendar melhorias para aplicações Android foi avaliada no contexto de três dispositivos diferentes. Foi possível obter uma redução de consumo de energia de até 14.73%.

Palavras-chave: CECOTool. CT+. Framework de Coleções Java. Perfil de energia.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADB	Android Debug Bridge
AOT	Ahead-Of-Time
API	Application Programming Interface
ART	Android Runtime
AST	Abstract Syntax Tree
CECOTool	Collections Energy Consumption Optimization Tool
CPU	Central Processing Unit
DAQ	Data Acquisition
GC	Garbage Collection
HAL	Hardware Abstraction Layer
IDE	Integrated Develpment Environment
JCF	Java Collections Framework
JDK	Java Development Kit
JIT	Just-In-Time
jRAPL	Java Running Average Power Limit
MSR	Machine Specific Register
RAM	Random Access Memory
RAPL	Running Average Power Limit
SEEDs	Software Engineer's Energy-optimization Decision Support framework
WALA	T.J. Watson Libraries for Analysis

CONTENTS

1

INTRODUCTION

1.1 MOTIVATION

In the recent years, with the widespread usage of mobile devices, such as: tablets, smartphones and, more recently, smartwatches; and also with the growth of the number of data centers, in which companies like Amazon, Google and Microsoft are involved; the demand for energy-efficient machines has been increasing. For one part, we have users requesting devices with more battery capacity or requesting systems that are specialized in saving energy, and for another part we have companies interested in reducing the energy cost of their servers, and also interested in reducing the CO^2 that is produced as a result of the resources that are used to power their data centers. These concerns contribute to the relevance of the subject of energy consumption, making it the main topic of many recent studies.

Initially, many studies were focused on optimizations of the infrastructure where software systems run. To name some, Heller *et al.* (2010) comes up with a solution to dynamically adjust the switches and ports of a network of servers in a data center in order to save energy, satisfying the changing traffic loads; Kültürsay *et al.* (2013) studies the Spin-Transfer Torque Random Access Memory (RAM) as a replacement for the commonly used Dynamic RAM, as a means of saving energy; and Ding *et al.* (2013) analyzes the impact of the wireless signal strength on the energy consumption of mobile devices, improving the state of the art power model for WiFi and 3G by incorporating the signal strength factor to it.

More recently, software aspects, such as data structures and concurrency constructs (Lima *et al.*, 2016; Pinto *et al.*, 2016), code refactorings (Sahin *et al.*, 2014) and development approaches (Oliveira *et al.*, 2016), have emerged as the main subject of some of the new studies in the area of energy consumption. Lima *et al.* (2016) benchmarks 10 operations on 23 functional data structures for the Haskell programming language, finding out differences in the time and energy consumption ranging between 2% and 85%. It also studies and benchmarks the behavior of two of the Haskell's data sharing primitives, showing that, in concurrent contexts, the execution time cannot be relied as proxy for the energy consumption, finding differences in the energy consumption that can vary not only depending on the operation and on the sharing primitive, but also due to the context in which they are used, thus concluding that there is no overall winner. Manotas *et al.* (2014) creates a framework capable of automatically changing a set of implementations of a given Application Programming Interface (API) on a given program's test suite, benchmarking every change to find the configuration with the most efficient energy consumption. Sahin *et al.* (2014) analyses 197 refactored versions of 9 applications, measuring the impact of the modified versions on the energy consumption. The study finds out that the changes can either improve or decrease the energy efficiency, and that commonly used predictors for energy consumption, such as the runtime and the execution count, may not accurately predict the energy impacts of applying the refactorings. Oliveira *et al.* (2017) evaluates the energy footprint of three available development approaches for the ANDROID platform: JAVA, JAVASCRIPT and C/C++. The study selects 33 public benchmarks, from the ROSETTA CODE and THE COMPUTER LANGUAGE BENCHMARK GAME (**TCLBG**) repositories, 22 of them in two versions, in JAVA and in JAVASCRIPT, and the remaining benchmarks in three versions, using all the cited languages. It also re-engineers four open source applications, using a hybrid approach where the modified version would use either JAVA and JAVASCRIPT, or JAVA and C/C++. The research finds out that the JAVASCRIPT approach consumed less energy than the JAVA approach in 26 of the benchmarks, but when it comes to the re-engineered apps there is no overall winner.

These and other studies in the area shed light and help us understand how complex and challenging is the task of properly measuring and predicting the impacts of software aspects on the energy consumption of an application. We've seen that factors such as the programming language, the target platform, the API implementation, the development approach, the usage context, and other software aspects, all play an important role on the energy consumption of an application. It is, therefore, natural that many developers, although having extensive knowledge on a language, are still unsure on how to build energy efficient systems (Pinto & Castor, 2017; Pang *et al.*, 2016). For this reason, many of the studies share a common concern: there is a lack of tools to support developers in building energy efficient applications.

On this research, we propose a tool called CT+, which analyzes Java code statically, detecting energy-efficient collections and recommending replacements that are more efficient regarding the energy consumption. The tool is an extension of another tool, called CECOTool, developed in the previous work of de Araújo Neto (2016). In this research, we (i) add two popular sources of collections that implement the Java Collections Framework (JCF): the Eclipse Collections and the Apache Commons Collections; (ii) we include collections that are not safe for concurrent access, (iii) we benchmark more operations, distinguishing the different positions where an element can be removed, added or retrieved, in sequential collections, (iv) we extend the tool to also recommend for Android applications, (v) we improve the static analysis by making use of points-to analysis and finally (vi) we automate the process of applying recommendations. With these improvements we intend to answer two research questions (**RQs**):

- **RQ1**: Can CT+ reduce the energy consumption further when compared to the original tool?

- **RQ2**: Are recommendations device-independent?

We'll show how CT+ was able to reduce the energy consumption further when compared to CECOTool (de Araújo Neto, 2016). We reuse one of the environments used on the original study and also a low-end desktop environment close to the one used on the original study to attempt a fair comparison. We also show how CT+ succeeded in reducing the energy consumption of Android applications, being able to achieve up to 14.73% of reduction in one of the mobile benchmarks.

1.2 STRUCTURE OF THE WORK

In Chapter 2, we introduce the JCF collections and present some of the collections of the two other sources that also implement the JCF. We present the approach that led to the creation of Collections Energy Consumption Optimization Tool (CECOTool) and we introduce the static code analysis concepts that are needed to understand CT+.

In Chapter 3, we start by presenting the limitations of the original tool. We then detail the improvements and new features that together comprehend CT+. Finally, we present a schema with an overview of everything that CT+ has in comparison with CECOTool.

In Chapter 4, we explain our methodology, introduce the benchmarks we used on our experiments and which precautions we took to prevent noises on the experiments. We then show and discuss our results.

In the remaining chapters, 5 and 6, we present relevant related works and conclude this study.

2

BACKGROUND

In this chapter we introduce the main concepts that are needed to understand this study. We start by presenting the standard JAVA collections and some of the collections of the APACHE COMMONS COLLECTIONS and the ECLIPSE COLLECTIONS; we talk about energy measurement in general and in particular for mobile devices; we present the static code analysis concepts that are needed to understand CT+; and finally we present the approach that led to the creation of CECOTOOL, from which CT+ was based off;

2.1 COLLECTIONS

In this section, we introduce the JAVA collections API and present some of the its general purpose collections. We explain the three categories in which each of these collections are included and we also introduce two other libraries, the ECLIPSE COLLECTIONS and the APACHE COMMONS COLLECTIONS, along with some of the implementations that we are going to consider in this research.

2.1.1 The Java Collections Framework

The JCF is the set of standard JAVA collections implementations and interfaces. It has three main categories of collections that are exposed in the form of API: **Sets**, where elements are stored without order and without repetition; **Maps**, in which elements are stored based on key-value pairs and hashing, and keys are unique; **Lists**, where elements are stored sequentially and can be accessed through indexes.

For each of these categories, the JCF provides different implementations that serve for different purposes. For example, there is `ArrayList`, which stores data sequentially on the memory, just as a regular array does. And there is `LinkedList`, which, as the name suggests, stores the data the same way a linked list data structure does.

The JCF also includes collections that are safe for threads, commonly called synchronized collections, meaning that they can be shared by multiple threads and be free from synchronization problems. Some examples are: `Vector`, `ConcurrentHashMap` and `Con-`

`currentSkipListSet`. Additionally, the JCF also has a functionality to wrap any collection with a synchronized construct, making it thread-safe. It is accessible through the `java.util.Collections` class, invoking the methods: `synchronizedList`, `synchronizedMap` and `synchronizedSet`.

There are many studies exploring the performance and the bottlenecks of the JCF. For example, Costa *et al.* (2017) finds out, in a comparison between the JCF and six other libraries with alternative implementations of the JCF API, that any `List` alternative implementation has better performance than `LinkedList`, they also discover that two of the studied libraries have `Set` implementations with better performance than `HashSet` and that primitive collections can be faster than `ArrayList` up to four times. Hasan *et al.* (2016) compares the JCF with the TROVE[1] and the APACHE COMMONS COLLECTIONS libraries and finds out there is always an alternative implementation that consumes less energy than any JCF collection in at least one operation. Some of the JCF collections included in this study are the following:

`ArrayList`. It employs an internal array to store its data, providing constant time on operations such as: `get`, `set` and `isEmpty`. It grows as elements are inserted, but it also exposes a constructor and a method, `ensureCapacity`, to give developers the ability to allocate space for new elements.

`LinkedList` has a linked list as its internal data structure. It has the advantage of not needing to shift elements when new values are not added at the end, but it has the disadvantage of taking $O(n)$ time for accesses.

`Vector` is similar to `ArrayList` in the sense that every element can be accessed in constant time through an index, but it follows a different strategy when it comes to space. Besides growing, it can also shrink whenever elements are removed. The growth strategy is ruled by two parameters, `capacity` and `capacityIncrement`, that can be specified through its constructor. The user can set its `capacityIncrement` to dictate by how much it will grow when necessary. Also, it is a thread-safe collection.

`CopyOnWriteArrayList` has an `ArrayList` internally and it is thread-safe, but whenever a changing operation occurs (e.g.: removal or addition), it clones itself with the applied change. This strategy, according to its documentation, can be quite expensive. However, this policy makes it possible to avoid the use of synchronization when reading and traversing. An iteration, or a retrieval operation, will act on a list whose state is preserved, as any change that occurs happens in a copy of the original list.

`HashMap` is the most used general purpose `Map` implementation. It uses a hash table base internally and, differently from its synchronized counterpart, `Hashtable`, it accepts `null` values and `null` keys. It provides constant time for the retrievals and additions, given the internal hashtable is properly dispersed. Its traversal time is proportional to its capacity. It lets the developer define two important parameters on its constructor: the initial capacity and the load factor. The initial capacity is used to allocate a predefined amount of space initially. The

[1] https://bitbucket.org/trove4j/trove/src

load factor is a `float` number between 0 and 1 that dictates a threshold for growing the table. Whenever the number of added entries is equal to the load factor multiplied by the capacity, the internal hash table doubles in size and is then rebuilt.

`LinkedHashMap`. By default, it stores elements in the order they were inserted, and this order is to be expected when iterating over it. It uses a doubly-linked list internally for the inserted elements, hence the name. It also provides a third parameter in its constructor, besides the initial capacity and load factor, which gives the possibility of defining a different ordering for the elements. By default it is the insertion order, the other option is to follow the last recently used order.

`Hashtable` is a thread-safe version of `HashMap`. Its synchronization mechanism works by locking the whole table when an operation occurs. This makes it very inefficient when it is shared across a high number of threads.

`ConcurrentHashMap` is also a synchronized collection that works similar to `HashMap`, but it has the advantage of not locking the whole table when a thread invokes operations on it. It locks just a portion of its internal table, making the other parts of the table still available for other threads. Thus, it can be thought of as an improved version of `Hashtable`.

2.1.2 The Eclipse Collections

The ECLIPSE COLLECTIONS [2] is a set of alternative implementations of the JCF. It was created and originally maintained by the Goldman Sachs company to attend their needs. It started off as a private library, in 2004, but it was then published on GitHub in 2012. Its set of collections comes with improvements ranging from performance to memory footprint, and also includes features that are not present in the standard JCF, such as collections for primitive types, bidirectional maps, composite collections, among others.

For this research, we tried to select general purpose collections, staying close to the functionalities provided and most commonly used from the JCF. The collections are the following:

`FastList` is designed to be a replacement for the `ArrayList` class from the JCF, but without support for the `ConcurrentModificationException`. It provides direct access to its internal array of elements, something that is not possible when sub-classing `ArrayList`. When an instance of `FastList` is initialized with zero capacity, its internal array points to a shared static array of size zero, making it memory-efficient on initialization. It only initializes its own instance of an array when at least one element is inserted.

`UnifiedMap` is a map implementation that employs an array internally and that does not use hashes. Alternate slots of this internal array serve as the key-value pair. Its creators say this configuration is more cache friendly because consecutive memory addresses are cheaper to access than hash mapped indexes. Since key and value might have different types, we conducted an examination on its source code to find out how this approach was implemented. We discovered

[2] https://www.eclipse.org/collections/

that it employs an internal array of type `Object`, making it possible to store objects of different types.

`ConcurrentHashMap`, as the name suggests, is a general purpose synchronized hash map. Its documentation does not have a description of its features and in which points it is different from the `ConcurrentHashMap` from the JCF. However, an examination of its source code revealed that it uses an `AtomicReferenceArray` to store its elements, whilst the JCF implementation uses a private implementation of the `Set` interface for its keys, and a private implementation of the `Collection` interface for its values.

`UnifiedSet` is an implementation of the `Set` interface that also provides methods required by an alternate collection interface, called SMALLTALK COLLECTION PROTOCOL.

2.1.3 The Apache Commons Collections

The APACHE COMMONS COLLECTIONS is another alternative implementation of the JCF. Differently from the ECLIPSE COLLECTIONS, it was open source since its beginning. But its purpose is similar: it provides more data structures implementations, complementing the JCF already offers. Among its implementations, we chose:

`TreeList` is an implementation of the `List` interface that is optimized for insertions and removals anywhere on the list. Its documentation says it is designed to ensure that all insertions and removals have complexity $O(\log n)$.

`NodeCachingLinkedList` is an implementation of the `List` interface that stores a cache of nodes when elements are added, potentially avoiding memory allocation and garbage collection on lists that receive multiple additions and removals. Its documentation states that it is more suitable for long-lived lists where both additions and removals occurs, and that its performance might be worse if this is not the case.

`CursorableLinkedList` is a list implementation that was created with the goal of providing a collection which allows the underlying list and the list iterator to be modified at the same time. To this end, it exposes the methods `listIterator` and `cursor`. The documentation makes it clear that the regular `iterator` method from the `List` interface shouldn't be used.

`HashedMap` is a general purpose map that serves as an alternative to `HashMap`. The original motivation behind this map was to provide the functionality of the class `MapIterator` that did not exist on Java Development Kit (JDK)1.7.

`StaticBucketMap` is a thread-safe implementation of the `Map` interface, designed for intense concurrent modifications. Its documentation states that it provides efficient retrievals, removals and additions, given the number of elements does not exceed the number of buckets - or slots - on the map. As the name suggests, the number of slots for this map is fixed at the time of the creation. Therefore, it is up to the developer the job of allocating enough buckets for the operations that are going to be necessary. Also, this map contains a monitor for each allocated bucket, meaning that concurrent accesses do not lock the entire entity, only a bucket. Because of this, it has the downside of not behaving as expected when multiple threads try to use the methods `putAll` or `removeAll`. The operation can be entirely canceled, leaving the map unchanged, or the result might be a mix of both operations.

2.2 MEASURING ENERGY CONSUMPTION

In this section we introduce the tools we used to measure the energy consumption of the benchmarks of this study. We start by presenting Running Average Power Limit (RAPL) and the jRAPL, and later the ANDROID POWER PROFILER.

2.2.1 Running Average Power Limiting (RAPL)

The RAPL is a mechanism capable of measuring and controlling the power consumption of the CPU, RAM and of other components, such as the level-three cache and the GPU. It was created by Intel and was presented by David *et al.* (2010). Modern Intel Central Processing Units (CPUs) come equipped with the RAPL interface, making Machine Specific Registers (MSRs) available for the developers. From these registers, developers can retrieve the energy consumption information. The RAPL interface is, unfortunately as for now, only available for LINUX platforms and MAC, through a kernel driver, and is only available on processors with architecture greater than or equal to Sandy/Ivy Bridge. As we use it for our measurements, the benchmarks for desktop platforms on this research are always ran on Linux environments. RAPL gives us the energy consumption information in four levels:

- Package: the total energy consumption on the CPU socket
- PP0: the total energy consumption of the CPU cores
- PP1: the total energy consumption of the components around the core (L3 cache, GPU, connectors)
- DRAM: the total energy consumption of the RAM

2.2.2 jRAPL

The jRAPL open-source JAVA library serves as an interface for gathering data from the machine-specific registers that come with RAPL compatible processors. It was developed by Liu *et al.* (2015) and eases the process of communicating with the RAPL interface. By employing it, JAVA programmers can easily measure the energy consumption of blocks, or even lines, of codes, without the need for any peripheral measurement device.

The communication with RAPL is intermediated by a JAVA class with native method calls. The methods that are called reside in a compiled `.so` file, which in turn have access to the

```
184     double[] energyBefore = EnergyCheckUtils.getEnergyStats();
185     final long start = System.currentTimeMillis();
186
187     startIteration();
188     try {
189       iterate(size);
190     } finally {
191       stopIteration();
192     }
193
194     final long duration = System.currentTimeMillis() - start;
195     double[] energyAfter = EnergyCheckUtils.getEnergyStats();
```

Figure 1: Example of the usage of jRAPL. Note lines 184 and 195.

MSRs. Lines 184 and 195 of the code on Figure 1 show how energy information can be retrieved using jRAPL. The call to `EnergyCheckUtils.getEnergyStats` returns a double array with the energy consumption information, as explained in Section 2.2.1.

2.2.3 Android

The ANDROID platform is one of the most popular mobile device platform on the market. It had 85% of the market share of the smartphones' operating system, in the 1st quarter of 2018, according to Statista[3]. Developers can write applications for it using JAVA and alternatively using JAVASCRIPT, C/C++ or KOTLIN.

Its operating system is open-source and Linux-based, composed of a stack containing Java APIs, C/C++ libraries, the Android Runtime (ART), system apps, a Linux kernel, a Hardware Abstraction Layer (HAL), used for the peripherals such as camera, and a power management module.

Being based on the Linux kernel makes it easier for manufactures to write drivers for it. The HAL provides a set of libraries through which JAVA applications can communicate with the device peripherals. These libraries are loaded on-demand, whenever an app makes a call to one of the libraries from the HAL.

The ART, in its current state, runs each app separately on its own process, within its own instance of the ART. The type of executable the runtime runs is called DEX. It is a type of bytecode specially designed to have low memory footprint. It features Ahead-Of-Time (AOT) and Just-In-Time (JIT) compiling, optimized Garbage Collection (GC) and debugging capabilities.

All the features available in Android for the developers are exposed through the Java API framework. Some features include: the resource manager, which makes it possible for the developer to access graphics, strings and layout files; the notification manager, making it possible

[3] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

for the user to display alerts; the activity manager, allowing the developer to configure the lifecycle of an app, among others. Some C/C++ libraries are also accessible for the developer through the Java APIs. One example is the graphics library, accessible from the `android.opengl` package.

Finally, system apps are also part of the stack. They are composed of support applications that come with the device from the factory. They include messaging apps, calendars, alarms, browsers, contact lists, and more. They also provide interfaces on which the developer can rely to build their apps.

2.2.3.1 The ADB

The ADB[4] is a command-line tool that is bundled within the Android SDK Platform tools. It makes it possible for developers to issue commands through a Unix shell to Android devices connected to a host machine. It is composed of three parts: a client, a server and a daemon. The client runs on the developer's machine. Whenever a command is issued, it searches for an active server and, if not found, a new server is instantiated. The server then searches for any device connected to the machine and establishes connections with them. The daemon is a background process that runs on the device and executes the commands that are issued to it.

From the adb, the user can query for connected devices, dump logs or battery usage information, open ports on the devices so they can be connected through Wi-Fi, among other things. In this research, we use the ANDROID POWER PROFILER (explained in Section 2.2.3.2), via ADB to gather the battery usage information, avoiding the burden of having to connect each experimental device to a Data Acquisition (DAQ).

2.2.3.2 Energy measurement in Android

In order to measure the energy consumption of ANDROID devices, researchers have been using two main approaches. One of them is to wire the device to a data acquisition system (**DAQ**), and the other approach is to use the ANDROID POWER PROFILER, which will be introduced below.

Using a hardware-based approach for measurement can be expensive and tiring, since each device has to be opened so we can wire them to a DAQ. Some devices can also be hard to wire. In particular, many medium and high-end contemporary smartphones do not allow their batteries to be removed in a straightforward manner. Thus trying to wire these devices can be time consuming and, given the number of devices that will be subjected to an experiment, it can be unpractical.

The ANDROID POWER PROFILER is accessible through the ADB using the command-line `adb shell dumpsys batterystats`. This command will output the battery usage

[4]https://developer.android.com/studio/command-line/adb

information since the last time the device was charged, including voltage level, overall and per-app energy consumption in milli-amper-hour (mAh) units, foreground time of each app, among others. The use of this profiler for energy measurements has the advantage of not requiring any wiring. The only downside to which researches naturally pay attention is that it may be not as accurate as hardware-based energy measurement tools. On this matter, (Nucci *et al.*, 2017) compares the two approaches, analyzing 54 ANDROID apps, showing that the margin of error between hardware and software based measurements is less than 5%. Thus, using the ANDROID POWER PROFILER presents calculated risks to the research.

2.3 STATIC CODE ANALYSIS

In this section we present the general concepts of static code analysis and we also go in-depth into the fundamentals needed to understand the improvements that were done in this research.

2.3.1 General concepts

Static code analysis is a type of software analysis that acts upon the software's source code, which can be in the form of compiled code or not. It is the opposite of dynamic analysis, where a running application is analysed. It has been used since the 70's to optimize compilers (Moller, 2018). More recently, it has been used to solve a range of other problems, such as: finding bugs in applications (Bessey *et al.*, 2010), security checks (Larochelle & Evans, 2002) and malware detection (Schmidt *et al.*, 2009).

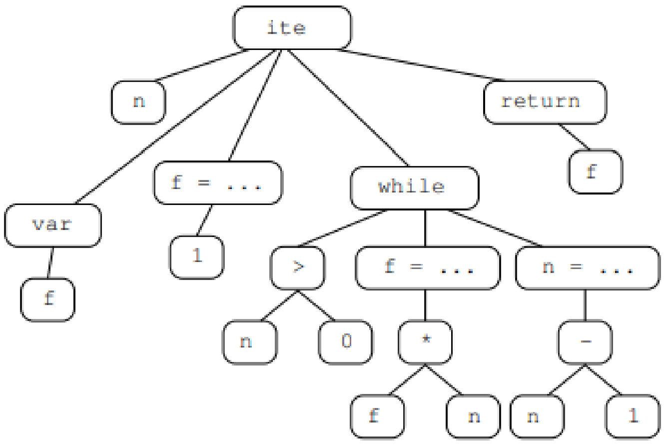
Usually, only a subset of the static code analysis concepts is used depending on the objective of the developer. For this reason, whenever a concept is introduced, the problem that the given concept tries to solve is mentioned along with it. Here, we focus on the main concepts that help us understand our tool. We first talk about the two data structures that are used to represent source code's statements:

Listing 2.1: `ite` function

```
public static int ite(int n) {
    int f;
    f = 1;
    while(n>0) {
        f = f*n;
        n = n-1;
    }
    return f;
}
```

Abstract syntax trees (ASTs), which is also used by compilers, is a form representation where the order of execution of the statements does not matter. In this representation, the statements are child nodes of functions. It is useful when the analysis does not need to take into account the program flow (Moller, 2018). For instance, given the Listing 2.1, its AST is represented on Figure 2. This is the representation used on points-to analysis, which will be explained in Section 2.3.2.

Figure 2: Ite function AST, taken from (Moller, 2018, p.11)



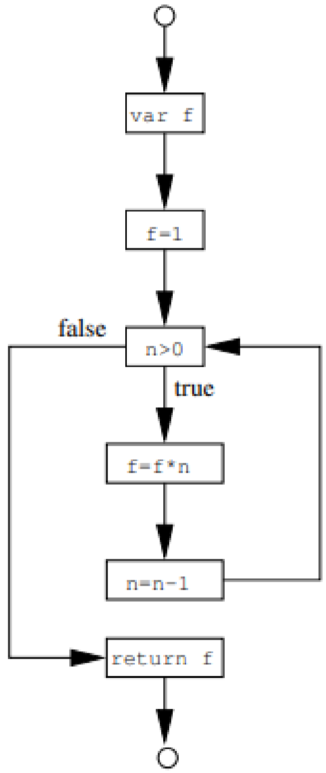
Control flow graphs (CFGs) are used in analysis where the order of execution of the statements matter. This representation is a directed graph where the nodes correspond to statements and the edges correspond to possible flows. The CFG representation of the Listing 2.1 is shown in Figure 3.

Regarding the analysis that use these structures, we start by describing the INTERPROCEDURAL ANALYSIS, which is an analysis that uses a CFG and, therefore, takes into account the order of execution of a program's statements. It receives this name because it is able to consider multiple methods in its analysis. It works by first constructing the CFGs for all the individual methods of a program; then it treats function calls by using two nodes, a call node with an edge going from the original method to the callee, and an after-call node pointing back to the caller where the execution must resume. This analysis can still vary regarding its sensitivity to context. We say that the interprocedural analysis is context-insensitive when it does not distinguish between different calls to the same function. The counterpart, when it does distinguishes the different calls to a method, is described in Moller (2018). But it is often not used on a whole program, due to being expensive, it is usually used together with heuristics to consider only parts of a program.

2.3.2 Points-to Analysis

Points-to analysis, or pointer analysis, is described in Moller (2018). It is a concept of static program analysis that aims to find to which objects the pointers in a program point. It makes use of an strategy, called allocation-site abstraction, to deal with the fact that there is no

Figure 3: Ite function CFG, taken from (Moller, 2018, p.13)



heap information available without running a program. This concept creates, for each allocation instruction found in code, an unique index to an abstract memory location. In this manner, every pointer has, as a target, an abstract memory location.

The analysis result can vary regarding its sensitiveness to flow or context. The flow-insensitive pointer analysis is often used due to being computationally cheaper than its sensitive counterpart (Moller, 2018˙ p.107). The result of this analysis is, for each pointer, a set of possible variables to which the pointers may point during the program execution. The flow-sensitive analysis outputs a different set for each program location where a pointer assignment occurs. The context sensitiveness is related to taking into account the method, and its call origin, when defining the points-to set of a pointer. For example, if a pointer is passed to the same method as a parameter in two different locations, a context sensitive analysis would have two different points-to sets for that pointer on that method.

2.4 T.J. WATSON LIBRARIES FOR ANALYSIS

The T.J. Watson Libraries for Analysis (WALA) library is a static analysis library that is able to read Java bytecode, Dalvik bytecode and JavaScript. Its set of features include: construction of class hierarchies, interprocedural analysis, context-sensitive analysis, points-to analysis, call graph construction, among others.

In this research, we make use of points-to analysis in CT+ to consider the different methods in which a collection is being used and only recommend if the recommendation is the same for all of these methods.

Regarding this topic, WALA's built-in points-to analysis is flow-insensitive, and the context-sensitiveness can be controlled through 2 entities, the `HeapModel` and the `ContextSelector`. The HeapModel dictates how instantiated objects will be disambiguated. For example, if `String` objects do not matter in an analysis, the `HeapModel` can be adjusted so that all the `String` allocations will point to the same allocation site. The `ContextSelector` dictates the context rules that will decide if a method will be cloned as a result of different calls. For example, one way to decide if a method will be cloned is by using the call-string approach (Moller, 2018, p.82), which will produce different method contexts for each different point in code that calls that method. This approach also defines a constraint, which is the maximum level of methods in the call stack allowed in the analysis.

WALA comes with 3 default analysis policies, which define different context-sensitiveness. The first policy is the most cheaper, called `ZeroCFA`, which creates just one allocation site for each object type in the code, and uses a single context for each method, meaning it is context-insensitive. The second policy is called `ZeroOneCFA`, which creates one allocation site per allocation found in code. But also creates only a single global context for each method. And finally, there is the `ZeroOneContainerCFA` which offers object-sensitivity for collection objects, meaning that every position in a collection gets its own allocation site, but this makes the analysis more expensive.

This research uses the `ZeroOneCFA`, which is sufficient for our needs. With it, we aim to gather the points-to set for collection type variables, and from that we get all the other pointers pointing to the same variable, called aliases, giving all the possible variables that may point to a same collection in memory.

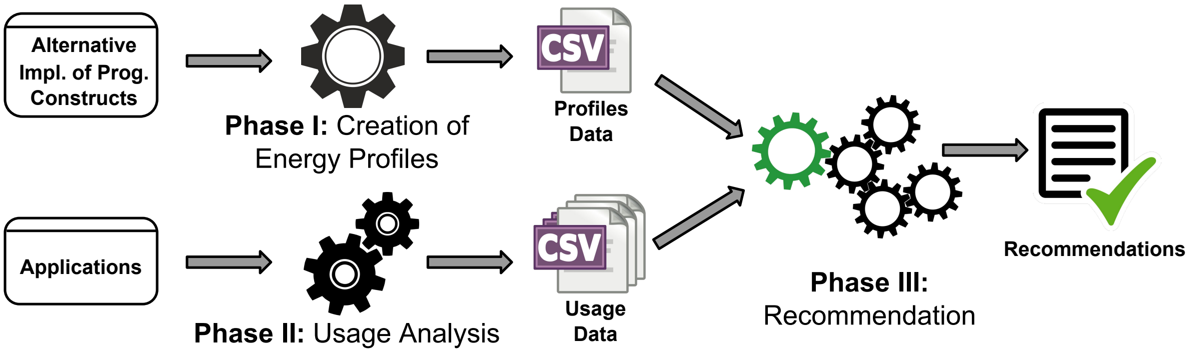
2.5 CECOTOOL

The work of de Araújo Neto (2016) proposes an approach to recommend changes on pre-defined interchangeable software abstractions, such as implementations of an interface, classes, data structures - among others - according to the energy profile of each possible abstraction implementation. This approach is depicted in Figure 4.

Phase I: Creation of energy profiles, which is the part of the tool in charge of measuring the different energy consumptions of each abstraction implementation. It follows strict rules elucidated by Georges *et al.* (2007), to avoid garbage collection and JIT compiler influence. The output of the profiler is also directed to the recommender, which will use it on its formula to decide which abstraction implementation is the best for each identified hotspot.

Phase II: Usage analysis, which receives as input the compiled code of the target application. It searches the code for energy variation hotspots, which are places that can easily

Figure 4: CECOTool flow



be changed to improve the energy consumption of the target application. On its analysis, it takes into account the nesting loop level, if any, of an operation, and also carries out this information interprocedurally. The output of the analyzer is a metadata file that will be later used by the recommender, with indications of where and how intense an abstraction operation is being used.

Phase III: Recommendation, which is the last end of the approach. It uses the data from the analyzer and the profiler to estimate the different energy consumptions on each different energy variation hotspot. For this, de Araújo Neto (2016) uses a formula that takes into account the number of occurrences and the nesting level of an operation: $totalFactor = \sum p_i \times u_i + \sum p_p \times u_p + \sum p_r \times u_r$
In which p_x means the consumption of a specified operation (x is i for insertion, p for traversal and r for removal).

CECOTool is an instantiation of the proposed approach, targeting thread-safe JAVA collections from the JCF. In this instantiation, de Araújo Neto (2016) was able to reduce the energy consumption of two real-world highly concurrent applications: XALAN, reducing the energy consumption by up to 3.49% and TOMCAT by up to 4.37%.

3

THE DEVELOPMENT OF CT+

In this chapter, we explain how we built CT+. We start by giving an explanation about the original solution and then discuss the limitations that led us to extend it, building a new tool capable of covering those limitations and that also includes features from other related studies. In Section 3.14 we build a detailed schema of the original tool and of CT+, highlighting the differences between them. In Section 3.13 we talk about the limitations of CT+.

3.1 CECOTOOL

The original tool was developed with the intent of instantiating the recommendation approach explained in Section 2.5. Thread-safe collections from the JCF was one possible choice of many other kinds of abstractions to which the approach could be applied. The reasons for choosing collections came from the importance they have on applications. They are extensively used when writing software and are usually focus of performance improvements (Xu, 2013; Costa & Andrzejak, 2018). Choosing the wrong collection for a task may even cause applications to consume too much memory or CPU (Costa *et al.*, 2017). CECOTOOL achieves good results by reducing the energy consumption of two real-world multi-threaded applications by up to 4.37%. Therefore, it proves that the proposed approach works.

3.2 LIMITATIONS OF THE CECOTOOL

The tool, as originally proposed, targets only thread-safe collections from the JCF. Also, it targets only desktop environments and profiles 9 operations, three operations from each of the three JCF interfaces. Additionally, the positioning of the operations on sequential collections are not distinguished, which we know to have impact on the energy consumption (Hasan *et al.*, 2016).

In this research, we intend to focus on recommending collections to reduce the energy consumption of applications. We take, for this end, CECOTOOL as a starting point. We add more collections to the tool, including the non-thread-safe collections of the JCF, and we also add collections from two other sources that implement the JCF; we distinguish and profile the different positions of the operations on sequential collections; we improve the static analysis of the tool; and we also make it compatible with ANDROID devices, giving us a new platform to explore. We put together these and other improvements, building a new tool which we called CT+.

3.3 IMPROVEMENTS

In this section we explain and detail all the improvements that together comprehend CT+. We start by summarizing them on Table 1. We then proceed to explain each of them individually.

Table 1: Improvements summary

Improvement	Description
Thread safety	JCF non-thread safe collections were included
More collections	Addition of the ECLIPSE COLLECTIONS and the APACHE COMMONS COLLECTIONS
Positioning of operations on sequential collections	We profile and consider the different positions where additions, removals and accesses happen on a list. We also check if they are being traversed sequentially or not
Better analysis	We use points-to analysis to find recommendations that attend all the methods in which a collection is used and we traverse every class to search for which collection instance is being assigned to a variable
Multiple recommendations for the same variable	To give the user more options, we output not only the best recommendation, but multiple energy-ordered options that are better than the original collection
Recommendations are automatically applied	Creation of a new module, the CT+ Transformer
Compatibility check before recommending a collection	We check if the behavior and the constructor being used on the original collection are compatible with the recommended collection
Android compatibility	We designed CT+ to be compatible with Android devices
The tool became IDE-independent	We made the tool IDE-independent by creating a command-line interface for it

3.4 THREAD SAFETY

The original tool by de Araújo Neto (2016) was focused only on thread-safe collections. A quick code search on GitHub[1] for the JAVA language, show us that non-thread-safe collections are more present on JAVA projects than thread-safe collections. The Table 2, with GitHub data from July/2018, gives us an overview. Also, previous researches about the JCF include the non-thread safe collections (Costa *et al.*, 2017; Hasan *et al.*, 2016). Besides being more popular,

[1]https://github.com/search

non-thread-safe collections also tend to be more efficient, since primitives to implement both pessimistic and optimistic concurrency are expensive from a performance standpoint.

Table 2: Code occurrences of Java collections July/2018

Collection	Is it thread safe?	Amount of ocurrences
java.util.ArrayList	No	28,602,937
java.util.HashMap	No	14,518,657
java.util.HashSet	No	5,597,900
java.util.Vector	Yes	4,192,029
java.util.LinkedList	No	3,331,905
java.util.Hashtable	Yes	1,768,504
java.util.ConcurrentHashMap	Yes	920,201

To correctly recommend between thread-safe and non-thread-safe collections, we consider the thread-safety of the original collection instance being assigned to a variable. The recommender was modified to ensure that thread-safe collections are only recommended when the original implementation is also thread-safe.

3.5 MORE COLLECTIONS

Another improvement of CT+, in comparison to the original CECOTOOL, is the use of sources of collections that are not part of the JCF. Hasan *et al.* (2016) analyzes the APACHE COMMONS COLLECTIONS and the TROVE library, Costa *et al.* (2017) analyzes a total of 5 other alternative implementations of the JCF, showing that there is always an alternative implementation that outperforms the standard Java collections, either in performance or in memory footprint. Thus, we added the ECLIPSE COLLECTIONS and the APACHE COMMONS COLLECTIONS to CT+, the reason being their popularity. By searching for their root packages - `org.eclipse.collections` and `org.apache.commons.collections` - on GITHUB, we found 466,394 and 1,022,778 code occurrences in Java projects, in January of 2019. A summary of all the collections included in this study, is shown on Table 3.

3.6 POSITIONING OF OPERATIONS ON SEQUENTIAL STRUCTURES

As evidenced by Hasan *et al.* (2016), taking the position of an operation on a sequential collection into account is important, as it can have strong impact on the performance and on the energy consumption of the operation. Hasan *et al.* (2016), for instance, finds out that `ArrayList` consumes less energy than `LinkedList` for elements added in the middle of the list, but `LinkedList` is the winner when it comes to insertions in the beginning. In our study, we corroborate these results and find more cases in which this happens. We extend CECOTool to distinguish between insertion and removals on the beginning, on the middle and on the end of a list.

Table 3: The selected implementations to be used in the CT+. Three different sources were used: Java Collections Framework, Eclipse Collections and Apache Commons Collections

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

The heuristic we used for this was to consider insertions or removals on the index zero as operations in the beginning. This heuristic works when a constant value is used as index, as we cannot infer with certainty when a variable is evaluated to zero statically; or when the method `addFirst()` of `LinkedList` is used. For operations in the middle, we consider that this is the case whenever a variable is used as index. We could have a false positive in this case, but due to the existence of operations that are used to explicitly add at the end or at the beginning, and due to the common use of variables as loop counters, the chances of a false positive are low. And as for operations at the end, we consider that this is the case when the method `add()`, which adds at the end, is used.

Table 4: Operations used on each collection.

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

We also distinguish between sequential or random access on a list. This is also inspired by Hasan *et al.* (2016) findings, that notices there is a difference on the energy consumption while traversing a list sequentially or randomly. The heuristic we used for this was to consider a list access as sequential whenever a variable being used as index is also being used on the tail of a loop in which the list resides, otherwise the access will be considered random. We carefully adjusted the **profiler** to not take into account the random number generation when calculating the energy consumption of accessing a list randomly. A summary of all operations we take into account is shown in Table 4.

3.7 BETTER ANALYSIS

During our initial experiments we were confronted with a problem when running some of the benchmarks. The energy consumption after some recommendations, along with the application performance, sometimes increased significantly. We found out that some of the recommendations made by CECOTOOL, when a collection is passed from a method A to another method B, were different for each method. This is a problem for two reasons: (i) method B does not know the type of the collection since the type of the parameter is often of interface type (e.g. `List`); and (ii) the recommendations for method B do not account for uses of the collection made by method A and these uses could produce different recommendations. The case is illustrated below.

```
class A {
    public static methodA() {
        //LinkedList would be recommended here
        //due to addition in the beginning
        List<Integer> list = new ArrayList<Integer>();
        for(int i = 0; i < 1000; i++){
            list.add(0,i);
        }
        B.methodB(list);
    }
}
class B{
    public static methodB(List<Integer> list){
        //ArrayList would be recommended here
        //due to traversal using index (mostly access in the middle)
        for(int i = 0; i < 1000; i++){
            list.get(i);
        }
    }
}
```

Cases like this one occurred in our experiments. We dealt with them by considering the different recommendations for the methods to which a collection is being passed, and only making a general recommendation applying to all these methods, if all the methods, when analyzed in isolation would receive the same recommendation. In this manner, we mitigate the risk of doing a recommendation that can have negative effects on other scopes. To do this, we recurred to points-to-analysis, using the WALA library, which is already employed on CECOTool.

The analysis used in this case was flow-insensitive, which means that instead of having different points-to sets for different parts of the program, the set will contain all possible pointed variables, and the `HeapModel` we used was able to distinguish different memory allocations. This is enough for us, as we only need to distinguish pointers that may point to the same variable.

Finally, with these parameters configured, we extracted all the pointers pointing to each member, static or local variable in which the type of the variable is `List`, `Map` or `Set`. CT+ outputs this new metadata on a `json` file in the end of the analysis. The **recommender** receives this file as input and, before recommending, checks if, for a given variable and its pointers, the recommendation in all the possible contexts are the same. The execution of the points-to analysis and the use of its file by the **recommender** were implemented as optional steps, as this analysis can be time consuming.

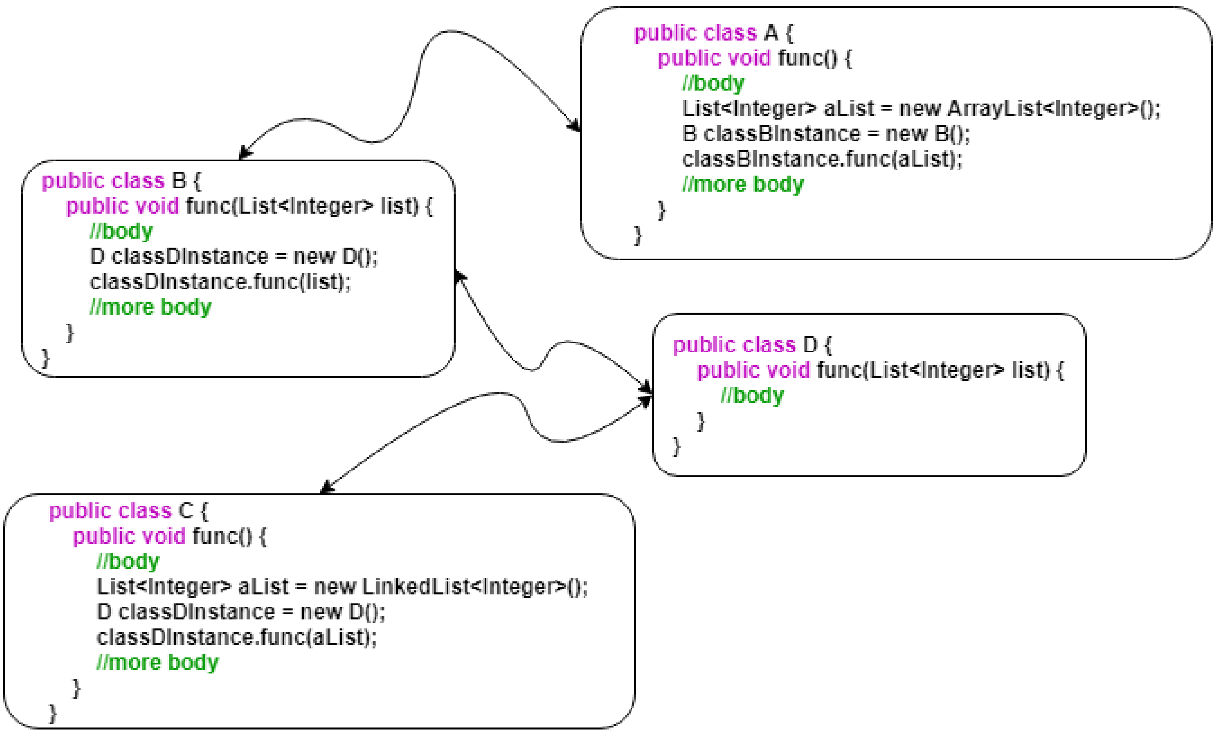


Figure 5: Example of the points-to analysis metadata

The points-to analysis output is a bidirectional graph where the nodes are variables of collection type along with the class and method where they are located, and the edges are pointers to other variables which may receive the same object instance as that variable. This graph is depicted on Figure 5. Whenever the recommender encounters a variable that is a node with at least one edge on that graph, it executes a depth-first search traversal on all connected nodes, and only recommends a collection when the recommendation is the same for all the visited nodes. For simplicity, in this process we compare the first recommendation of the ordered recommendations for each node. On the example shown on Figure 5, if a recommendation happens, it would require changing the list instances assigned to the variables `aList` of classes `A` and `C` to an instance that would be the same for both variables and that would also attend classes `B` and `C`.

The tool originally relied only on WALA to infer the variables type. The type inference API of WALA, from the `TypeInference` class, is intraprocedural. As a result, most of the types were inferred as being of the interface type: `List`, `Set` or `Map`. The drawback of this is that we do not know if the collection that is being assigned to a variable is thread-safe or not, and if we can indeed recommend a better collection or not. To mitigate this problem, complementing the WALA type inference, CT+ traverses every analyzed class' methods, constructors and declarations to discover which instances are being assigned to collection-type variables. The cases where an instance can not be found happen with variables that come from another method or class. Those cases are handled by the points-to analysis, as explained before.

3.8 MULTIPLE RECOMMENDATIONS FOR THE SAME VARIABLE

After adding more collections and using the tool extensively on real applications, we noticed that, sometimes, the collection with the lowest energy footprint was not compatible with the original collection. The reason being: constructor arguments that are not available on the recommended collection; or a change of behavior between the recommended and original collection. For this reason, CT+ recommendation file outputs all the possible replacements for the original collection, rather than just the best option, ordered from the lowest energy footprint to highest. An example is shown in Figure 6.

	A	B	C	D	E	F	G	H
1	Field name	Is local?	Source code	Containing class	A method that uses it	Original collection	Ordered recommendations	Choosen recommendation (used by the CECOTool transfomer, change it accordingly)
2	map	false	52	org.apache.commons.mat	getTransformer	java.util.HashMap	unifiedMap(EclipseCollections)<hashedMap(Apa	1
3	map	true	473	org.apache.commons.mat	applyTransform	java.util.HashMap	unifiedMap(EclipseCollections)	1
4	map	true	115	org.apache.commons.mat	applyTransform	java.util.HashMap	unifiedMap(EclipseCollections)	1

Figure 6: Multiple recommendations example. Part of the COMMONS MATH recommendations for **note**

3.9 CREATION OF A NEW MODULE, THE CT+ TRANSFORMER

While using the tool and applying the recommendations, we noticed that although the changes are easy to apply, a considerable amount of time is taken changing the source code. The problem with this is that, given the number of the recommendations, it can also be error-prone. To make this process faster we developed a new module for CT+ to be in charge of applying the recommendations to the source code, which we called the CT+ Transformer[2].

This part of the tool, differently from the analyzer, receives the project's source code and the recommendations file as input, which can be manually changed. For example, the developer can change the recommended collection by changing the last column of the file, shown in Figure 6, or he can also delete lines from it to avoid a recommendation. This module works by using the JavaParser[3] library, which parses the code and makes it possible to use declaration and assignment visitors to apply the recommendations.

3.10 COMPATIBILITY CHECK BEFORE RECOMMENDING A COLLECTION

After applying recommendations and trying to compile an application for the first time, it was not rare to run into compilation problems originated from constructor mismatch. This mostly happened with lists. For example, it happens when an `ArrayList` collection that is instantiated with initial size is replaced by `LinkedList` or `TreeList`, which do not have a constructor where the developer can inform the initial size. In order to avoid this problem, CT+ checks if the constructor used on the original collection also exists on the recommended collection, if it doesn't exist, it then recurs to the next best collection available. For this to work we visited each collection documentation and hard coded which constructors are available for them, the **recommender** uses this information and proceeds as explained.

Another compatibility check we understood was important was the collection's behavior check. For example, `LinkedHashMap` maintains the order in which an element is inserted and this order is expected to be the iteration order, whereas `HashMap` does not have this behavior. So making a substitution from `LinkedHashMap` to `HashMap` is risky, but not the opposite. Similarly, `TreeMap` stores the elements according to the natural ordering[4] of its elements, making it risky to change from `TreeMap` to `HashMap` or `LinkedHashMap`. Thus, we checked the collection's documentation and implemented this additional check on CT+, recurring to the next best recommended collection when a substitution is not possible due to behavior change.

3.11 ANDROID COMPATIBILITY

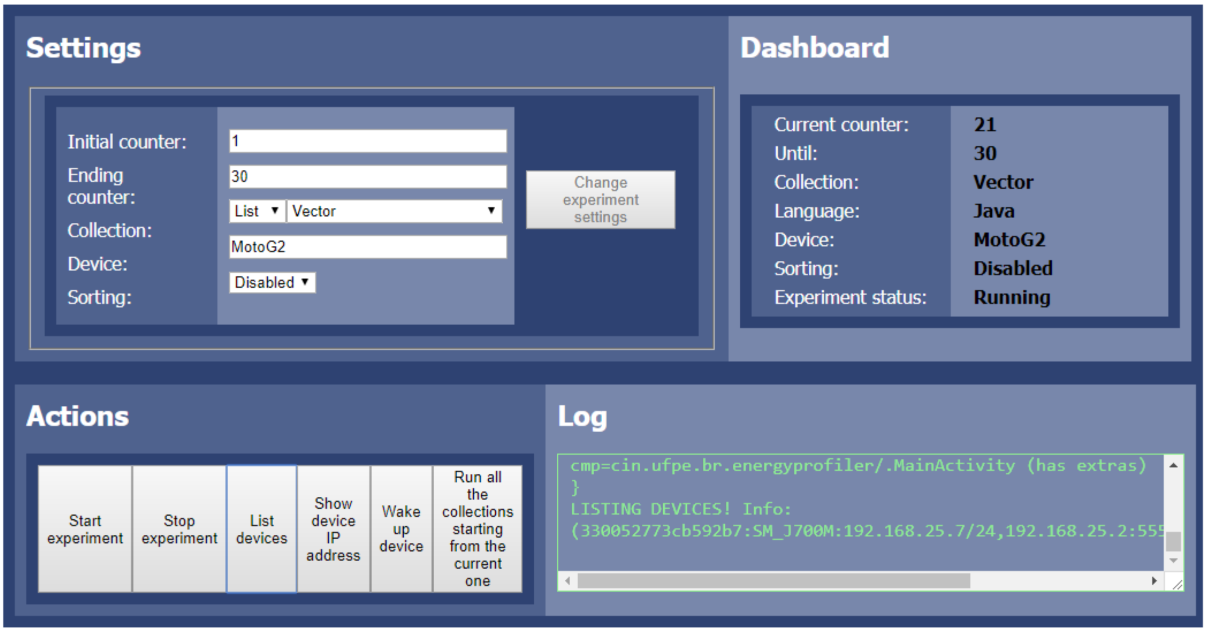
Although we decided to do our measurements with the ANDROID POWER PROFILER, which already saves us from having to connect our devices to DAQ systems, we still had the

[2]https://github.com/ros3cin/CTplus-transformer
[3]http://javaparser.org/
[4]The order dictated by an object's **comparator**

burden of having to recompile our apps whenever we tweaked something or changed benchmarks. Thus, to save us from this, we built a web dashboard, with NodeJS technology.

The dashboard can be instantiated multiple times, allowing us to benchmark multiple devices at once, showing us their respective experiment statuses. It also provides actions to make it possible for us to stop a benchmark, start from any point in the experiment, list the attached devices and run all the benchmarks available in sequence. The dashboard interface can be seen in picture 7.

Figure 7: Picture of the dashboard



In order to use the ECLIPSE COLLECTIONS on the Android devices, we had to use a JDK7 compatible version of it. The reason being we have only at most Android API 23 with us, which is at most compatible with JDK7. We also tried to make our profiling and benchmarking applications compatible with most of the Android devices by targeting the minimum API as being 15. The Android API version compatibility distribution, from July 2018, is shown on Table 5.

3.12 MAKING THE TOOL IDE-INDEPENDENT

The original tool, by de Araújo Neto (2016), requires the user to setup the project on the SCALA IDE FOR ECLIPSE[5] in order to use it. This process can be time consuming and may prevent new users, which do not have expertise with the IDE or the language, from trying the tool. It also makes machines such as servers, which usually do not employ a graphical user interface (GUI), being unable to run it. Furthermore, a project setup on a IDE is a step that should only be required by developers interested in contributing to the project. For this reason,

[5]http://scala-ide.org/

Table 5: Android Platform version cumulative distribution (July/2018)

Android Version	Version Name	Api Level	Cumulative distribution
4.0	Ice Cream Sandwich	15	99.99%
4.1	Jelly Bean	16	99.2%
4.2	Jelly Bean	17	96.0%
4.3	Jelly Bean	18	91.4%
4.4	KitKat	19	90.1%
5.0	Lollipop	21	71.3%
5.1	Lollipop	22	62.6%
6.0	Marshmallow	23	39.3%
7.0	Nougat	24	8.1%
7.1	Nougat	25	1.5%

we equipped CT+ Analyzer and Transformer modules with a command-line interface, and also made their compiled versions available in the form of `.jar` on their GitHub pages[6][7], in the releases section. For instance, the Analyzer command-line usage help is shown on Figure 8.

3.13 LIMITATIONS OF CT+

The improvements and features presented on the previous sections aim to not only make the quality and the scope of the recommendations better, some of them also target the usability of the tool and accelerate some of its processes, such as the addition of a command-line interface to make the tool IDE-independent and the creation of the Transformer module to automatically apply the recommendations. But still, there are some aspects of the tool, that will be discussed here, that need further improvement.

The profiling phase, specially for mobile devices, can take hours to complete. This aspect started to be problematic after the addition of more collections and after we started gathering the energy profiles for the mobile devices. Some collections are so different in their performance that a workload that is reasonable for one implementation may be too much for another implementation. One example of this is the difference between adding to `Vector` and adding to `CopyOnWriteArrayList`. The energy consumption along with the runtime performance is orders of magnitude different. This means that a workload that runs for 20 seconds on `Vector` can run for 2000 seconds on `CopyOnWriteArrayList`. We were able to manage this discrepancy on the desktop platform by using small workloads, since jRAPL is capable of fine-grained measurement. But this strategy was not possible on the mobile platform

[6]https://github.com/ros3cin/CTplus
[7]https://github.com/ros3cin/CTplus-Transformer

```
Usage: <main class> [-ahpr] [--analysis-output-file=<analysisOutputFile>]
                    [--energy-profile-file=<energyProfileFile>]
                    [--points-to-analysis-file=<pointsToAnalysisFile>]
                    [--recommendation-output-file=<recommendationOutputFile>]
                    [-e=<exclusions>] [-t=<target>]
                    [--packages=<packages>...]...
      --analysis-output-file=<analysisOutputFile>
                             The name of the analysis output file. Defaults to
                               analysis.csv
      --energy-profile-file=<energyProfileFile>
                             The energy profile file to be used on the recommender
      --packages=<packages>...
                             Space separated packages to include in the scope of the
                               analysis
      --points-to-analysis-file=<pointsToAnalysisFile>
                             The points-to-analysis output file
      --recommendation-output-file=<recommendationOutputFile>
                             The name of the recommendation output file. Defaults to
                               recommendations.csv
  -a, --analyze              Run the analysis
  -e, --exclusions-file=<exclusions>
                             The path to the scope exclusion file
  -h, --help                 Displays this help
  -p, --points-to-analysis   If set, runs the points-to-analysis on the target
  -r, --recommend            Run the recommendation
  -t, --target=<target>      The target JAR or APK, this is required if the analyze
                               flag is set
```

Figure 8: CT+ command-line usage help

because, as we will see on Section 4.1, we needed to make sure that every workload ran for at least 20 seconds.

Another aspect that can be improved is the compatibility of the tool with other desktop operational systems. Aside from needing Intel processors with architecture newer than or equal to Sandy/Ivy Bridge, the tool currently can only run its desktop profiler on Linux systems. This is a consequence of the convenience we have in Linux accessing the CPU registers that provide the energy consumption information. This access is intermediated by a kernel driver, called MSR, activated through the `modprobe` command, which jRAPL needs. Future work could contribute to this aspect by writing a driver for other operational systems to have access to the machine-specific registers on compatible Intel processors.

Finally, WALA's current version (1.4.3) is not able to read the names of local variables and is not able to accurately infer the source code line number of variables on Android applications. Hence, some of the recommendations for Android applications will have virtual variable names (numbers) and an approximation of the real source code line number. Thus, for the mobile applications, we had manually adjust the variables' names and their source code line numbers on the recommendations file. Should this issue be resolved in the future, we infer that an update of the library will be sufficient.

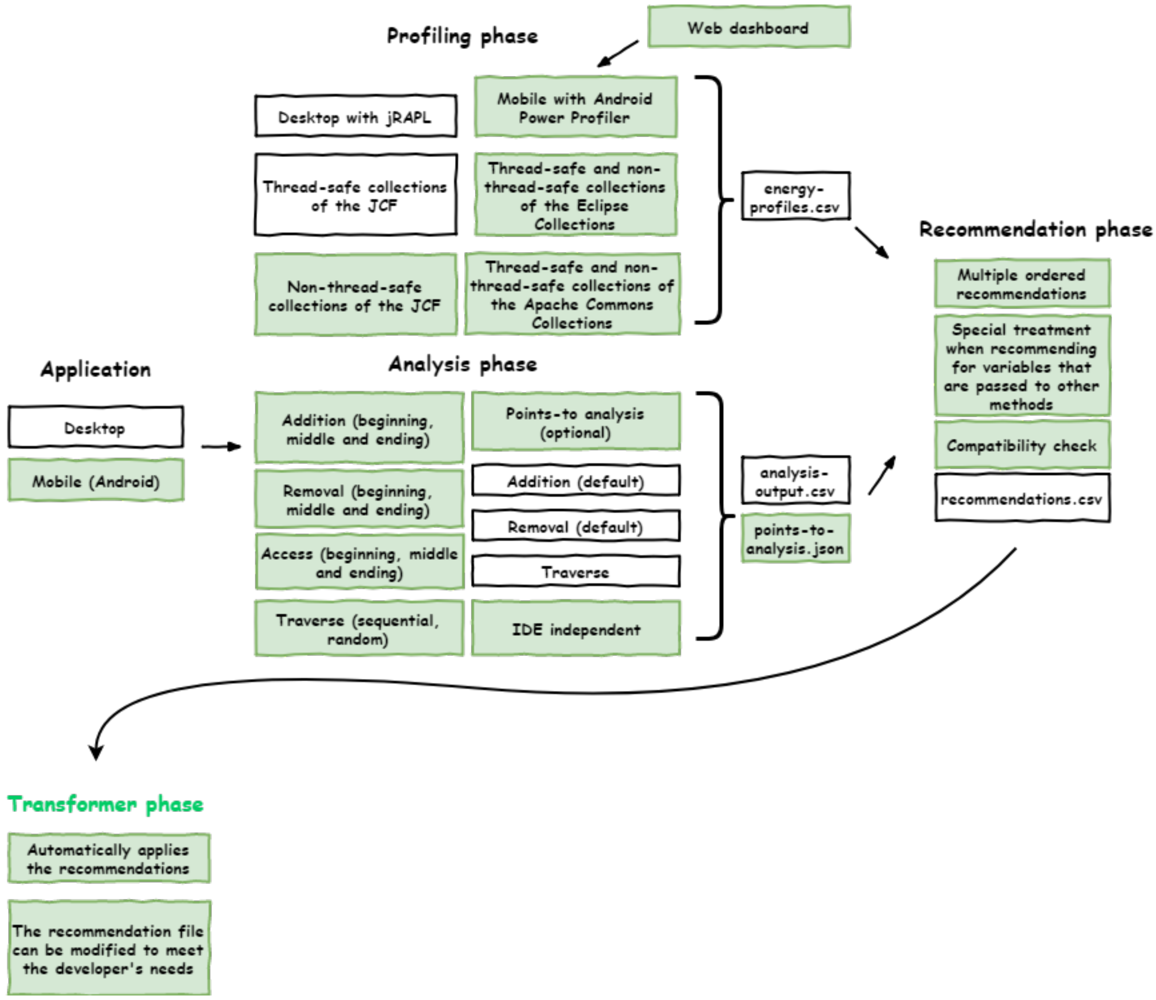


Figure 9: Schema comparison between CECOTOOL and CT+. The green boxes are the improvements of CT+. The white boxes represent the original tool features.

3.14 SUMMARY OF THE DIFFERENCES BETWEEN CECOTOOL AND CT+

In order to have an overview of the improvements and new features that CT+ implements, we show on Figure 9 a schema where the CT+ improvements and features, in green boxes, are put together with the aspects that were inherited from the original tool, presented in the white boxes. With this representation, we are able to see that CT+ improvements reach all the modules of the tool, bringing five new aspects to the profiling phase, six new aspects to the analysis phase and three new aspects to the recommendation phase. Each of these aspects can still be broken and detailed further. Additionally, the tool is able to do recommendations for mobile Android applications, and now possesses a new module - the Transformer module - capable of automatically applying the recommendations.

4

EVALUATION

In this chapter we present the evaluation of CT+. In this evaluation, we aim to assess wether CT+ can be used to reduce the energy consumption of real-world systems, and also to quantify the impact of the proposed improvements, answering the following research questions (RQs):

- **RQ1**: Can CT+ reduce the energy consumption further when compared to the original tool?
- **RQ2**: Are recommendations device-independent?

This chapter is organized as follows: the **methodology** section, in which we present the devices and benchmarks we used in our experiments, explaining how we measured the energy consumption of each and what precautions we took to reduce any threats to the experiments and which workloads we used; the **benchmarks** section, in which we present all the benchmarks we used; the **results** section, where we begin by looking into the energy consumption reductions achieved by CT+ on the benchmarks, followed by the analysis of the recommendations that were applied; the **discussion** section, where we discuss the results, talking about dominance between collections and sharing our insights about the achieved results; the **threats to validity** section, where we expose the threats to validity of this study and what we did to mitigate them.

4.1 METHODOLOGY

To evaluate CT+ we use three types of execution environment: **server**, **desktop** and **mobile**. These environments differ in processing power, available memory, use of battery and measurement procedure. We employ jRAPL to perform the energy measurements on the server and on the desktop environments, and for the mobile environment we use the Android Energy Profiler. We use five different devices, summarized on Table 6. As desktop environment we use a notebook (**note**) with an Intel Core i7-7500U with four 2.7GHz cores, and 16GB of RAM. For the server environment we use a high-end server (**server**) with a two-node Intel Xeon E5-2660 v2 processor with 20 2.20GHz cores (10 per node) and 256GB of RAM. As for the mobile

Table 6: The devices used in the experiments and their characteristics

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

environments, we use three smartphones: a Samsung Galaxy J7 (**J7**), a Samsung Galaxy S8 (**S8**), and a Motorola G2 (**G2**).

When creating the profiles for each environment, we took the same precautions of the previous works of Oliveira *et al.* (2017) and Hasan *et al.* (2016). We ran a series of micro-benchmarks, 30 times each, comprised of pairs of collection-operation, from which we collected the energy consumptions and used the average value. We reduced the influence of the JIT (Georges *et al.*, 2007) by running a warm-up phase, comprised of 10% of the workload of the benchmark, before collecting the samples. Whenever profiling thread-safe collections, we used four threads, dividing the workload between them, and when profiling for the non-thread-safe collections, we used just one thread.

We analyzed seven desktop-based benchmarks: BARBECUE, BATTLECRY, JODATIME, TWFBPLAYER, XISEMELE, XALAN and TOMCAT; two mobile-based: FASTSEARCH and PASSWORDGENERATOR; and three on both-environments: APACHE COMMONS MATH 3.4, GOOGLE GSON and XSTREAM. These benchmarks, with the exception of FASTSEARCH and PASSWORDGENERATOR, were all used in previous related work (de Araújo Neto, 2016; Pereira *et al.*, 2018; Hasan *et al.*, 2016). XALAN and TOMCAT, specifically, were used in the original work of de Araújo Neto (2016), from which we built CT+. So the results of these two benchmarks will be our main reference of how much further we reduced the energy consumption, making them important subjects of our experiments.

BARBECUE, BATTLECRY, JODATIME, TWFBPLAYER and XISEMELE are used to assess a tool developed on the work of Pereira *et al.* (2018), called JSTANLEY, which also uses static code analysis to recommend energy-efficient collections. We use them to check how well CT+ was able to reduce their energy consumption when compared to JSTANLEY. APACHE COMMONS MATH 3.4, GOOGLE GSON and XSTREAM are used on the work of Hasan *et al.* (2016), where the original collections being used on these benchmarks are replaced to reduce or increase the energy consumption, given their energy footprints.

For TOMCAT, we were not able to use all the libraries that were included on this research. We had to exclude the ECLIPSE COLLECTIONS, the reason being the DACAPO BENCHMARK SUITE endorses the use of JDK 6 for the TOMCAT benchmark to work properly, which is not compatible with ECLIPSE COLLECTIONS. Thus, we could only use the APACHE COMMONS

COLLECTIONS, which is compatible with JDK 6.

As for the workloads, for TOMCAT and XALAN, we used the workloads provided by the DACAPO SUITE. The only difference is the number of threads that the suite spawns when executing the benchmarks. On **server**, the number of threads is 40, on **note** the number of threads is four. For BARBECUE, JODATIME, TWFBPLAYER and XISEMELE we used the test suite that comes with them. For BATTLECRY, we use as benchmark a class inside the app that was designed to test it. These are the same approaches used on the work of Pereira *et al.* (2018). For GOOGLE GSON and XSTREAM, we constructed a class to exercise each Java primitive. Since these are serialization libraries, the benchmark consisted of serializing this class. For APACHE COMMONS MATH 3.4, we executed multiple statistical functions from its API. As for both PASSGENERATOR and FASTSEARCH, since they are very simple apps, designed for a very specific functionality, their workloads consisted of executing their main function (e.g., generating passwords).

The mobile devices required extra care when executing the benchmarks. Whereas the jRAPL is capable of fine-grained energy measurement, the Android Energy Profiler collects this information at process level. Therefore, in order to mitigate any noise or imprecision, we adjusted the workloads so that they could run for at least 20 seconds.

For the majority of the experiments, we collected the results of 30 executions of each benchmark of both original and modified version, for each device. The only exception was the TOMCAT benchmark, which we executed both versions 600 times and discarded the first 30 executions. This was needed because, according to the DACAPO SUITE developers, it has a very flat warm-up curve[1] when compared to the other benchmarks of the suite. As for the results comparison, since most of our samples were not normally distributed, according to the Shapiro-Wilk's normality test (S. Shapiro & B. Wilk, 1965), we used the Wilcoxon-Mann-Whitney test (Wilks, 2011) to test wether the differences on the energy consumption were statistically significant. We also employ the Cliff's Delta (Cliff, 1993) as a measure of effect size. We did not remove outliers.

4.2 BENCHMARKS

In this section, we present all the benchmarks used in this study along with the suite or repository from which they were taken. We organize the sections by suite or repository name, and in each of them we to introduce the benchmarks we used. A summary with all the benchmarks we use on the experiments, along with their source and target platform is presented on Table 7.

[1]Section 4.2 of https://github.com/dacapobench/dacapobench/blob/master/benchmarks/RELEASE_NOTES.txt

Table 7: All the benchmarks used on the experiments

Benchmark	Source	Platform
TomCat	DaCapo Benchmark Suite	Desktop
Xalan	DaCapo Benchmark Suite	Desktop
Barbecue	SourceForge	Desktop
Battlecry	SourceForge	Desktop
JodaTime	SourceForge	Desktop
Xisemele	SourceForge	Desktop
Twfbplayer	SourceForge	Desktop
Google Gson	GitHub	Desktop/Mobile
XStream	GitHub	Desktop/Mobile
Apache Commons Math 3	GitHub	Desktop/Mobile
Fast App Search	F-Droid	Mobile
Password Generator	F-Droid	Mobile

4.2.1 Benchmarks from the DACAPO BENCHMARK SUITE

The DACAPO BENCHMARK SUITE[2] (Blackburn *et al.*, 2006) is a collection of benchmarks for the JAVA programming language, featuring real applications with non-trivial behavior. The set of benchmarks it includes are: ARVORA, BATIK, ECLIPSE, FOP, H2, JYTHON, LUINDEX, LUSEARCH, PMD, SUNFLOW, TOMCAT, TRADEBEANS, TRADESOAP and XALAN.

In this research, we use XALAN and TOMCAT to compare our results to the results of de Araújo Neto (2016). Additionally, we modify the suite by adding jRAPL to it and making it output the energy consumption of an execution of a benchmark, along with the performance data it already outputs.

Following the advice of the suite developers, we report that the suite version we use in this study is the version **9.12**. We highlight that although the version of the suite is different from the one used by de Araújo Neto (2016), the benchmarks are the same. We used this new version because it fixes a number of repository URLs - needed to build the benchmarks - that, due to being outdated, were broken or no more existent. Also, similarly to the aforementioned work, we use the large workload size of XALAN and TOMCAT benchmarks. In our experiments, we use these two benchmarks, explained below:

XALAN, is an application that processes eXtensible Stylesheet Language for Transformation (**XSLT**), being able to transform XML documents into HTML, text or other types of XML documents. It can be ran as a standalone application, from the command-line, or from inside another application, in the form of library.

APACHE TOMCAT is a well known Java web server that, in the version used in this study (**6**), implements the JAVA SERVLET and JAVASERVER PAGES specifications from the JAVA

[2]http://dacapobench.org/

COMMUNITY PROCESS[3]. It can be used to either deploy web applications or web services.

4.2.2 Benchmarks from F-DROID

F-DROID[4] is a catalog, available online and also as an ANDROID app, of free and open source applications for the ANDROID platform. We chose two applications from the catalog and exercised the core method of each to use as benchmarks. They are:

FAST APP SEARCH TOOL, which we call FASTSEARCH for short, is an ANDROID app that helps users finding apps, making it possible to search them by package name. We

PASSWORD GENERATOR is an app capable of generating random passwords for the user. These passwords are safely stored by one master password defined by the user.

4.2.3 Benchmarks from GITHUB

GITHUB[5] is an online repository of open source code, founded in 2007, that runs the GIT source code version control. We chose three applications from there that were also used in the related work of Hasan *et al.* (2016). They are described below:

GSON is a JAVA library, created by GOOGLE, that is capable of serializing JAVA classes into `json` and deserialize back to a class. It doesn't require the developer to put any kind of annotation on a class or on its attributes to make it serializable. Additionally, it is also capable of deserializing classes, that are in the form of JSON, whose source code is not available on the application that is deserializing. To use this library as a benchmark, we built a class containing, as attributes, JAVA primitives, such as `int`, `boolean`, `String`, `float`, `double` and a `List` of integers initialized with a variable number of elements, depending on the desired workload.

XSTREAM is an open source library that serializes JAVA classes into XML and back. Similar to GSON, it also doesn't require the use of annotations, making it easy to use. Also, due to its similarities with GSON, which involve serializing and deserializing classes, we used the same approach as in GSON to use it as a benchmark.

COMMONS MATH 3 is an open source library, developed by APACHE, which adds support to mathematical and statistical functionalities that are not available in the JDK. It addresses common mathematical and statistical problems, such as solving a linear system of equations, generating random vectors of data, hypothesis tests, among others. The benchmark we built for it involved exercising its API using a set of examples found on its documentation and on its unit tests.

[3] https://www.jcp.org/en/home/index
[4] https://f-droid.org/en/
[5] https://github.com/

4.2.4 Benchmarks from SOURCE FORGE

SOURCE FORGE[6] is an online repository of both open source and proprietary software. For the applications taken from there, we use their unit tests as benchmarks, except for the BATTLECRY application, in which we used a test input, that comes within the app, as benchmark. The approach we chose to follow for these benchmarks are the same used on the related work of Pereira *et al.* (2018). A description of each benchmark is given below:

BARBECUE is an open-source JAVA library capable of creating and displaying bar-codes.

BATTLECRY is an open-source application that generates lyrics for songs using a list of words and grammar definitions, both provided by the user.

JODA TIME is a library designed to replace JDK's date and time classes, including full support for other types of calendar, such as the gregorian calendar and the buddhist calendar.

THE WEST FORTBATTLE PLAYER, TWFBPLAYER for short, is an open source application that replays battles from a browser game called THE WEST[7].

XISEMELE is an open source library for JAVA that makes it possible to read, edit and write XML documents.

4.3 RESULTS

On this section we divide our results into two groups: **desktop and server**, and **mobile**. For each group, we first look into the energy reduction achieved in each benchmark, and later we discuss the recommendations that were applied.

4.3.1 Desktop and server results

From Table 8 we can see the results of applying CT+ to the **server** and **desktop** environments. The column **improv** shows how much more energy the original version of the benchmark consumed when compared to the modified version. A positive value indicates that the modified version consumes less energy than the original version. For the **server** environment, only TOMCAT and XALAN were executed, as these are applications that are expected to be executed on a server. The TWFBPLAYER and XISEMELE benchmarks had no statistically significant difference between the original and modified version, for this reason they are not shown on the table. As for the remaining benchmarks, CT+ was able to reduce their energy consumption. Also, according to Romano *et al.* (2006), which says that a Cliff's Delta greater than 0.474 is considered large, CT+ recommendations resulted in versions with large effect size. For XALAN, in particular, the effect size was 1, meaning that every execution of the modified version of the benchmark had lower energy consumption than the original. JODATIME exhibited the greatest reduction, with a value of 6.65%.

[6]https://sourceforge.net/
[7]https://www.the-west.net/

Table 8: Results for the desktop and server environments. Energy results are red for the original versions and green for the modified versions.

Device	Benchmark	Improv	Changes	p-value	Mean(J)	Stdev	Effect Size
note	Barbecue	4.37%	21	7.0^{-4}	56.17 53.71	2.70 2.53	0.50
	Battlecry	2.82%	4	1.5^{-3}	67.95 66.06	2.67 3.18	0.48
	Gson	0.7%	16	8.0^{-5}	29.93 29.72	0.22 0.16	0.57
	Commons Math	1.02%	133	6.3^{-12}	48.93 48.43	0.29 0.15	0.90
	JodaTime	6.65%	16	$< 2.2^{-16}$	123.02 114.83	2.42 3.50	0.94
	Tomcat	3.96%	13	$< 2.2^{-16}$	32.77 31.47	1.02 0.41	0.86
	Xalan	4.77%	63	$< 2.2^{-16}$	107.04 101.93	0.19 0.15	1
	Xstream	2.53%	95	3.122^{-13}	59.97 58.45	0.52 0.49	0.94
server	Tomcat	4.83%	60	$< 2.2^{-16}$	89.33 85.01	2.06 2.03	0.86
	Xalan	5.49%	56	$< 2.2^{-16}$	242.29 228.98	4.4 7.02	0.86

Table 10 shows which collections were replaced, according to the recommendations of CT+. In both **server** and **note**, we can see that `Hashtable` was substituted by `ConcurrentHashMapEC` many times on XALAN benchmark, 49 times on **server** and 48 times on **note**. Also, commonly used collections from the JCF, such as `ArrayList`, `HashMap` and `Vector`, were also replaced by alternative, more efficient, collections from the ECLIPSE COLLECTIONS and from the APACHE COMMONS COLLECTIONS. TOMCAT recommendations varied a lot between both environments. Whereas **note** had 13 recommendations, **server** had 60 recommendations. Most of the recommendations of the latter were to replace `HashMap` by `HashedMap` (from APACHE COMMONS COLLECTIONS), which happened 39 times. As for the former, most of the recommendations were to replace `Hashtable` by `ConcurrentHashMap`, which happened six times. It is important to reiterate that we were not able to use the ECLIPSE COLLECTIONS for the TOMCAT recommendations, for reasons explained in Section 4.1. As for the other six benchmarks, there were 285 recommendations. Only three of these recommendations suggested the use of collections from the JCF. 88 of the recommendations suggested the use of collections from APACHE COMMONS COLLECTIONS, and 194 from ECLIPSE COLLECTIONS. Once again it is possible to observe a trend of replacing well-known collections such as `Hashtable`, `HashMap`, and `ArrayList` by more energy-efficient but

Table 9: Results for the mobile environment. Energy results are red for the original versions and green for the modified versions.

Device	Benchmark	Improv	Changes	p-value	Mean(J)	Stdev	Effect Size
S8	Commons Math	10.16%	26	1.25^{-8}	92.06 82.70	2.59 9.61	0.86
	FastSearch	0.085%	5	1.67^{-3}	35.06 35.03	3.32 1.78	-0.47
	Google Gson	0.97%	11	6.42^{-4}	16.45 16.29	0.22 0.20	0.40
	PasswordGen	4.44%	2	2.38^{-9}	16.86 16.11	0.41 0.65	0.90
J7	Commons Math	-0.33%	22	2^{-4}	23.82 23.90	2.33 2.62	-0.56
	Google Gson	4.78%	9	3.2^{-3}	13.78 13.12	1.59 2.67	0.44
	PasswordGen	14.73%	5	6.44^{-9}	12.83 10.94	0.90 0.76	0.87
G2	Commons Math	-1.16%	27	0.0091	17.22 17.42	0.51 0.14	-0.41

less-known alternatives.

4.3.2 Mobile results

From Table 9, we can see that the mobile results varied a lot among devices. For instance, CT+ recommendations for COMMONS MATH on **S8** had the second best energy reduction of the mobile devices, whereas the recommendations of the same benchmark for **G2** and **J7** resulted in versions that consumed more energy than the original versions. The best energy reduction was obtained on **J7**, where the original version of the PASSWORDGENERATOR benchmark consumed 14.73% more energy than the modified version. The reduction of this benchmark for **S8** was of 4.44%, more than 3 times less than on **J7**. FASTSEARCH recommendations resulted in a more efficient version only for **S8**, albeit small (0.085%). For **J7**, the energy consumption of FASTSEARCH was not statistically different from the original version. For **G2**, CT+ did not generate any recommendations for FASTSEARCH and PASSWORDGENERATOR, meaning that the tool estimated that the original collections being used on these two apps are already the best efficient alternative for **G2**.

The recommended collection for the mobile devices are summarized on Table 11. It is interesting to see that the COMMONS MATH benchmark on **S8** has more recommendations to replace the original collection to another JCF collection than all the benchmarks we evaluated on the **note** machine combined. On the one hand, the only collection recommended by CT+

that is not from the JCF for this benchmark 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`, or `FastList`, instead of `ArrayList`, and `LinkedHashMap` in place of `HashMap`. For the remaining benchmarks, CT+ made few recommendations, 11 for GSON, two for PASSWORDGENERATOR, and five for FASTSEARCH. Overall, the recommendations only produced a large effect size for COMMONS MATH and PASSWORDGENERATOR. Furthermore, these were the only benchmarks that could achieve energy savings greater than 1% in the **S8**. Among the 22 recommendations of COMMONS MATH on **J7**, 14 were for ECLIPSE COLLECTIONS and eight were for APACHE COMMONS COLLECTIONS. In all these cases, CT+ recommended that developers replace `ArrayList` by an alternative implementation. For this specific context, the recommendations did not yield energy savings. CT+ also recommended replacing `ArrayList` by alternatives in the case of GSON and PASSWORDGENERATOR. These substitutions yielded considerable energy savings. The **G2** differed from the others in this study in the sense that only one of the benchmarks 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.

4.4 DISCUSSION

This section presents a more in-depth discussion about the results achieved in the previous sections.

4.4.1 Prevalence of the alternative implementations of the JCF

The implementations from ECLIPSE COLLECTIONS and APACHE COMMONS COLLECTIONS were the most recommended. On the desktop and server environments, out of the 477 recommendations, they were recommended 446 times, accounting for more than 93% of the recommendations on that environment. On the mobile environment, albeit not so frequent, they were still the majority of the recommendations. They were recommended 77 times, out of the 107 total recommendations for the mobile environment. Agregatting all the results, the collections from the JCF amount for only 11.47% of the recommendations.

4.4.2 Commonly used collections and energy efficiency

Our results imply that the most commonly used collection from the JCF have a more energy-efficient counterpart. 97.9% of all the statistically significant recommendations for the server and desktop environments that CT+ performed were replacements for `Hashtable` (121 times), `HashMap` (140 times), `HashSet` (20 times), `Vector` (8 times), and `ArrayList` (178 times). This results follows the popularity of these collections, shown in Table 2 of Section 3.4.

Since they are popular, it is expected they constitute many of the recommendations. Corroborating with the observation that they have a more energy-efficient counterpart, they were mostly not recommended as replacements. Exceptions ocurred when, for example, `LinkedList` was replaced by `ArrayList`, where traversals can be orders of magnitude more optimal. These results, along with the improvements on the energy consumption, suggest that these collections might not be good choices when energy consumption is important, raising the importance of considering alternative implementations when this is the case.

We investigated further and looked into the metadata generated by CT+ to elucidate why `ArrayList` was replaced so many times. The reason to the attention given to that collection is due the fact that it is arguably the most popular collection of the JAVA language. Two factors help explain the lack of recommendations in its favor and why it was replaced so often. First, the majority of the operations on sequential collections are the `add(value)`, which adds an element at the end of the collection, and `iteration(random)`. `ArrayList`'s energy footprint for both of these operations is, in most of the devices, greater than on `FastList`, which is an alternative general purpose implementation of the `List` interface, especially designed to be more efficient than `ArrayList` in terms of speed. Also, differently from `ArrayList`, it does not throw concurrent modification exceptions. Consequently, it is able to provide direct iterator access to the internal array of items [8]. Secondly, there are many cases where `ArrayList` is the most efficient collection, but since it is the most commonly used collection, chances are it is already being used, thus no replacement is recommended by CT+. This is what happened on FASTSEARCH and PASSWORDGENERATOR on **G2**. In other words, as a result of being widely used, in cases where `ArrayList` is the most efficient collection, it is already being employed and thus no more benefits can be achieved by changing it, given the collections' libraries included on CT+.

4.4.3 Different devices matter

Although for some cases, such as in the FASTSEARCH applications, the recommendations were similar. Our results show that, in general, they vary significantly across devices. For example, for XALAN on **note**, CT+ recommended that 10 `ArrayList` instances be changed to `FastList` and one to `NodeCachingLinkedList`. For **server**, in contrast, it recommended changing from `ArrayList` just two times, suggesting the use of `TreeList`. And in both machines, the energy consumption decreased.

The effectiveness of CT+ in decreasing the energy consumption also varied across devices. XSTREAM, for instance, for most of the devices, did not result in a version that significantly reduces the energy consumption. The only exception was on **note**, where CT+ was able to reduce its energy consumption significantly (p-value of 3.12^{-13}), and with a large effect size (0.94). We can attribute to this result the differences between recommendations and devices.

[8] https://www.eclipse.org/collections/

On **note**, CT+ applied 95 modifications, whereas the mobile device with most changes (**G2**), only had 41. The collections that were target of the recommendations were also different. On **note**, `ArrayList` was replaced by `FastList` 21 times, and by `LinkedList` one time. On **G2**, `ArrayList` was replaced by `TreeList` three times. Those two 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.

4.4.4 Number of recommendations and energy reduction

Contradicting our natural assumption that more recommendations imply more savings, our results suggest that there is no correlation between the number of recommendations and the energy reduction. For instance, **note** had 133 recommendations for the Commons Math application, achieving a reduction of 1.02% in the energy consumption, whereas for the JodaTime application, with only 16 recommendations, we achieved the reduction of 6.65%, the best energy saving for **note**. Doing a comparison between the number of recommendations for TomCat on **note** and **server**, we can see that while the former had only 13 recommendations, the latter had 60, but the difference between the percentages of energy reduction was of only 0.87%. Furthermore, for the mobile benchmarks, the number of recommendations for Commons Math on **S8**, **J7** and **G2** were 26, 22 and 27 respectively. These numbers are relatively close, taking into account the discrepancy found on the mentioned desktop cases. But the energy consumption reductions for each of those devices on that application were surprisingly different, whereas for **S8** we had the reduction of 10.16%, for the other two devices we had an increase on the energy consumption.

4.4.5 Dominance among collections implementations

Among all the 40 different collection implementations, CT+ only recommended 20 of them. When trying to understand this behavior, we noticed that some collections completely dominate (Peterson, 2009) the others. We say that a collection implementation dominates the other when, for every operation, subject to a device and a given workload, that collection always have lower energy footprint than the other. Since every dominated collection has a dominating alternative, they will never be recommended by CT+.

Figure 10 depicts dominance relation for the thread-safe `Map` implementations on the **server** machine. Based on this dominance relation, only four thread-safe `Map` implementations can be recommended by CT+ on the **server** machine: `ConcurrentHashMap`, `SynchronizedLinkedHashMap`, `ConcurrentHashMapEC` and `SynchronizedUnifiedMap`. These are collections that are not dominated by any other. For instance, as shown in the picture, `Hashtable` is dominated by `ConcurrentHashMapEC`, and `Hashtable` itself dominates `SynchronizedTreeMap`. Thus, `ConcurrentHashMapEC` also dominates `Synchro-`

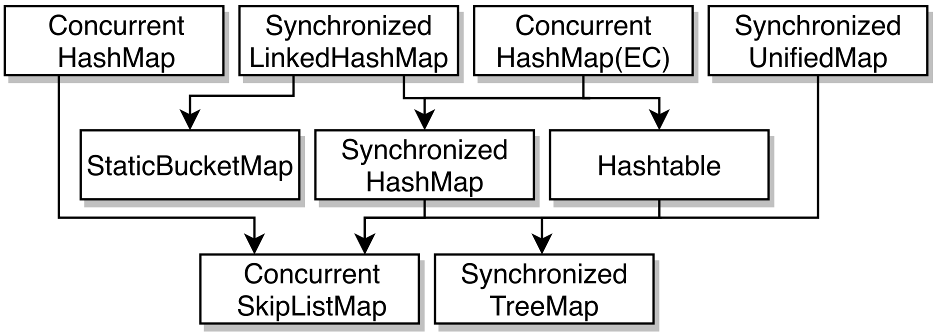


Figure 10: Order of dominance between the thread-safe Map implementations on **server**. Arrows point from the dominating collection to the dominated one.

`nizedTreeMap` transitively. Then, for the **server** device, `SynchronizedTreeMap` and `Hashtable` will never be recommended. In fact, we observed that `Hashtable` was dominated in every device we experimented with. This result, along with the scalability issues it presents, discussed in Pinto *et al.* (2016), and with the vast availability of more energy-efficient alternatives, suggests that it should be rarely used in practice. Implementations such as `ConcurrentSkipListSet`, `SynchronizedTreeMap` and `SynchronizedUnifiedMap` were dominated in three out of the five devices.

4.5 THREATS TO VALIDITY

Regarding **internal validity**, we mitigate the influence of other factors on the energy consumption by reducing the number of any background processes or applications that usually run along with the devices we used. Also, we deactivate energy consumption policies such as turning off the screen, on both mobile and desktop, and hibernation on desktops. On the mobile platform, we make sure that whenever we start running the 30 executions of each benchmark, the battery capacity is at 100%. This mitigates the influence of batteries with varying voltage on the energy consumption and this also gives each benchmark the same starting point, thus having a fair comparison of the consumed energy. Additionally, we always adjust the mobile workload so that each sample runs for at least 20 seconds, decreasing any imprecisions that the Android Energy Profiler may present. As for the statistical inferences, we used Shapiro-Wilk to verify wether the samples followed normal distribution and, in negative case, resorted to the use of the non-parametric test of Mann-Whitney-Wilcoxon, with 95% confidence level.

We highlight that, although we compare the differences on the results and on the recommendations between the desktop and mobile platforms, the tools used to measure the energy consumption of the benchmarks are different for each platform, presenting a potential threat to validity. Whereas we use jRAPL for the desktop platforms, which is capable of fine-grained measurement, on the mobile platform we use the Android Power Profiler, which outputs the energy consumption at process level. This can impact the measurement of the reductions achieved in each platform because at different levels of granularity we have different resources being taken into account.

For the **external validity**, we run our experiments in three different classes of devices, desktops, servers and smartphones, amounting to the total of five devices, achieving positive results on most of the benchmarks. Despite this, we can not generalize our results. We could see that, although not frequently, some of the recommendations had negative impact on the energy consumption. This can happen to any other pair of device-application that might be subject to our tool. Also, we highlight that it is necessary that the target application make medium to extensive use of collections for the changes to be significant. We limit our results to Java collections, as other languages, despite having the same data structure abstractions, may have different interfaces and implementations. Furthermore, we show how different the energy consumption reduction and the recommendations can be, depending on the device. For this conclusion, we use devices with notably different specifications. They vary in: the number of cores, processing power, amount of RAM, model, brand, among other specifications. Further investigation is needed to check if the same variability happens between different devices with equal specifications.

Table 10: Recommended collections for **note** and **server**

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

Table 11: Recommended collections for **S8**, **J7**, and **G2**

Benchmark	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

5
RELATED WORKS

Regarding energy consumption optimization, there is the interesting work of (Manotas *et al.*, 2014), in which the Software Engineer's Energy-optimization Decision Support framework (SEEDs) was created. It also aims to automate the entire process of optimizing the energy consumption of a software. But, differently from our work, for this to happen the application must've been previously prepared with test cases that can be tweaked in order for the algorithm to run its strategy.

The inputs of the framework are: the application code, a set of potential code changes, optimization parameters and context information. The potential code changes could be, for example, the different implementations of `List` from the JCF, the optimization parameters could be, in this case, the different consumption of each known `List` operation and the context information could be the platform in which the tool will be ran.

What is also interesting in this study, and that is directly linked with this work, is that an instantiation of SEEDs was done targeting the JCF and alternative collections, aiming to identify improvements from switching from a collection to another. They first built a preliminary study where 13 benchmarks were created and a collection from the JCF was initially chosen. They ran each benchmark 10 times, switching collections each time and then they counted how many times switching from the initial collection improved the energy consumption. In 7 of the benchmarks there were benefits from switching the initial collection, up to 96% improvement, and in 6 cases the energy consumption was negatively affected, and the increase in the consumption reached a value of up to 2,598%, which is evidence that it is easier to worsen an application's energy consumption than it is to make it better.

Later they went on to test their instantiation of SEEDs on 7 real applications. They made two experiments, one of which they only allowed SEEDs to switch between JCF implementations, and on the other one they allowed the framework to switch between any alternative implementation. From the results we can see that the improvement gain from allowing the tool to use other collections implementations was at max 3%. Thus, despite knowing that alternative JCF implementations can outperform the standard Java collections, the complexity of a real application makes it more challenging for a non-standard collection to make difference on the overall efficiency.

The work of Costa & Andrzejak (2018) presents the COLLECTIONSWITCH, an approach for switching Java collections at runtime, taking into account the collection allocation site and individual collection peculiarities, in this case performance and memory footprint. The study makes use of adaptive collections, which are collections that use other collections internally, allowing them to change due to pre-defined conditions. For example, the researchers at this work used the `AdaptiveSet`, from KOLOBOKE COLLECTIONS[1], which works internally as an `ArrayList` for small sizes and works as a `HashMap` for large sizes.

Their approach involve changing applications code so they can be aware of allocation site and workload. Adaptive collections variants are then introduced in place of the original collections so they can change between the desired variants at runtime. They also need, as we did on our research, to collect workload profile from the collections' operations that they are going to take into account, such as add, put, remove or iterate. They also come up with a formula to calculate the estimated cost of a variant replacement, similar to what (de Araújo Neto, 2016) had to do, that takes into account the number of operations, weighted by the workload used, and the average cost of that operation on a specific collection, for a specific maximum number of elements.

They evaluate their approach on a collection of microbenchmarks, the COLLECTIONS-BENCH[2], and also on 5 applications from the DaCapo Benchmark Suite. Their results show they were able to improve execution time on all types of collections abstractions: maps, sets and lists, when compared to the JCF standard collections; and they also managed o improve the execution time of the applications by up to 15% and reduce the peak memory usage by up to 10%.

These results corroborate the conclusion that there is no definite winner when it comes to collections. We saw that even for a specific operation there is a better alternative than the best implementation can offer, which is the possibility of using more than one implementation at once, with adaptive collections.

The work of (Hasan *et al.*, 2016) is another study about the Java collections energy consumptions from which some of the improvements presented in this research were based off. It analyzes the energy behavior of collections' implementations of the JCF, the Apache Commons Collections and the Trove library, and it targets the Android mobile platform.

The experiments are conducted in an infrastructure called GREENMINER, originated from (Hindle *et al.*, 2014). Similar to the dashboard presented in this research, it is composed of: a web server (the host), which store results, show information about running tests and provides control of the clients; and a client, which is composed of a Raspberry Pi attached to an Android device. In this case, the energy measurement is done by a sensor of current called Adafruit INA219[3].

Corroborating with our findings, the research also concludes that there is no winner

[1] https://koloboke.com/
[2] https://github.com/DiegoEliasCosta/CollectionsBench
[3] https://www.adafruit.com/product/904

among the different implementations. For example, it is shown that, among the studied libraries, `HashMap` was more energy efficient when mostly insertion and random access is required, but if insertion order should be preserved, the `LinkedMap` from the APACHE COMMONS COLLECTIONS is a better alternative.

It also shows how the different `List` implementations behave depending on the position of an insertion and depending on the type of list access (random or sequential). Due to these results, we made CT+ capable of distinguishing the different positions of insertions and removals. And we also came up with a heuristic to distinguish random from sequential accesses.

Pereira *et al.* (2018) build a tool called jStanley with the same purpose of CT+, it recommends Java collections with the purpose of saving energy. It is a Eclipse plugin which traverses the Abstract Syntax Tree (AST) finding direct invocations of collections' methods and also indirect invocations, which is defined as an invocation of a method containing a direct invocation of a collection operation. It then exposes for the user, in the form of a warning on the Integrated Develpment Environment (IDE), suggestions for replacing the current collection, along with an estimation of how much would be saved with the change. It also lets the user change the workload size and also prioritize performance to change the suggestions. The tool was able to reduce the energy consumption of the studied applications by up to 17%.

Taking a look on the applied recommendations[4], we realized that the tool allows the substitution of thread-safe collections to non-thread-safe collections and vice-versa. On CT+, since we want to maintain the target code's original behavior and also due to different workload and number of threads used on the profiling phase, which makes it not viable for us to compare the consumptions of collections with different thread-safeness, we do not allow this kind of change. But this work make us think of an interesting further improvement of the tool which would aim to investigate if it is safe to change collections with different thread-safeness.

Helano *et al.* (2015) develop a tool called ECODROID, which works in the form of ANDROID STUDIO plugin, capable of identifying parts of the application code of ANDROID apps that may result in an anomalous energy consumption. The tool uses a model for estimating the energy consumption of the multiple components of a ANDROID device, such as GPS, WIFI, AUDIO CARD and CPU. This model is an adaptation of a model developed in the previous work of Couto *et al.* (2014), which calibrates the consumption of each of the mentioned components for a ANDROID device. ECODROID works by first automatically instrumenting, using the JAVAPARSER library, the target code that is of interest for the analysis. In this process, the original code is cloned with new lines added to either the test and the application classes. The instrumented code is then updated, with the `android update project` command, and then a battery of tests is executed using the `adb shell am instrument` command. After the tests execution, their output is downloaded to the computer running the ANDROID STUDIO, with the `adb pull` command, and, with this data, ECODROID constructs a SUNBURST graph, that is displayed on the IDE. With this approach, although no rigorous

[4]https://github.com/greensoftwarelab/jStanley/blob/master/paper-resources/projectchanges.txt

evaluation of the tool was done, they state that the tool was being actively used by a research group that is in contact with big companies of the mobile area, having identifying, during its usage, that the `dispatchMessage(android.os.Message)` method from the ANDROID framework was the reason for many of the anomalies they found. They then show how they were able to successfully add an anomalous method to an existing application by making a call to `dispatchMessage(android.os.Message)` on that method's body. This work shows a different way to find energy variation hotspots, as explained in Section 2.5. While CT+ approach executes static analysis, giving weights for method calls inside loops, finding the usages of a target API and finally applying the results to a formula, ECODROID instruments methods, that will have its calls to a previously calibrated target API tracked and properly accounted, using a consumption model that depends on a device's components.

6

CONCLUSION

In this work, we implemented a series of improvements that were inspired by previous related studies with collections. We accounted for the impacts of the positioning of operations in sequential collections (Hasan *et al.*, 2016); we included two different sources of collections, both popular in GITHUB and also used in previous work (Costa & Andrzejak, 2018), and discussed how dominant they are over the standard JCF collections; we included non-thread-safe collections, hence covering the most popular collections of the JCF (e.g.: `ArrayList`, `HashMap`), where they accounted for the majority of the recommendations.

We improved the recommendation further by using points-to analysis so collections that are passed along to other methods are only recommended when the recommendation is the same for all methods. We also built CT+ to be compatible with the ANDROID platform, being able to successfully reduce the energy consumption of mobile applications. Also, we automated the approach even more by creating the CT+ TRANSFORMER module, which made it possible to apply recommendations efficiently, avoiding the error-prone task of applying them manually.

With these improvements, CT+ was able to further reduce the energy consumption on both benchmarks used on the original study, achieving the reduction of 5.49% on XALAN, and 4.83% on TOMCAT. We also saw significant energy consumption reduction on mobile applications, reaching up to 14.73% of reduction on PASSWORDGENERATOR and having positive results for most of the remaining pairs of benchmark-device.

These results answer our **RQ1**, in which CT+, along with all the new features it possesses, was able to further reduce the energy consumption of the two benchmarks used on the original study of de Araújo Neto (2016). They also helped us answering **RQ2**, where we could see how different the recommendations and the results are, depending on the device executing a benchmark.

Although we did our best to cover in CT+ features that would help replacing JAVA collections for more energy-efficient alternatives - consequently reducing the energy consumption of applications - we understand that we are still far from having the ideal tool. Future work could, for instance, include even more collections or even more operations; make use of the points-to analysis metadata to implement a different strategy to resolve the recommendations for collections that are shared among different methods; or improve the static analysis to take into account even more static code information.

Further investigation is needed to understand why some recommendations led to versions that consumed more energy than the original version. Also, the aspects of the studied collections that influence their energy profile still need to be elucidated. Understanding this phenomena and their interactions with the underlying hardware might be a challenging task, but we believe that this can be an important step towards generating energy-efficient software.

REFERENCES

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., & Wiedermann, B. (2006). The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114:494–509.

Costa, D. & Andrzejak, A. (2018). Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 16–26.

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*, 389–400.

Couto, M., Carção, T., Cunha, J., Fernandes, J., & Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. In *Programming Languages*, 77–91.

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

de Araújo Neto, J. B. F. (2016). UMA ABORDAGEM ESTÁTICA PARA RECOMENDAR ESTRUTURAS DE DADOS JAVA PARA MELHORAR O CONSUMO DE ENERGIA. Master's thesis, Federal University of Pernambuco, Brazil.

Ding, N., Wagner, D., Chen, X., Pathak, A., Hu, Y. C., & Rice, A. (2013). Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. *SIGMETRICS Perform. Eval. Rev.*, 41(1):29–40.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76.

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*, 225–236.

Helano, F., Rocha, L., & Gomes, D. (2015). Ecodroid: Uma ferramenta para análise e visualização de consumo de energia em aplicativos android. In *Conference: III Workshop on Software Visualization, Evolution, and Maintenance*.

Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., & McKeown, N. (2010). Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 17–17.

Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C., & Romansky, S. (2014). Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 12–21.

Kültürsay, E., Kandemir, M., Sivasubramaniam, A., & Mutlu, O. (2013). Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 256–267.

Larochelle, D. & Evans, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, 19:42–51.

Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., & Fernandes, J. P. (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)*, 1:517–528.

Liu, K., Pinto, G., & Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In Egyed, A. & Schaefer, I., editors, *Fundamental Approaches to Software Engineering*, 316–331.

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*, 503–514.

Moller, A. S. M. (2018). *Static Program Analysis*. Department of Computer Science, Aarhus University.

Nucci, D. D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., & Lucia, A. D. (2017). Software-based energy profiling of android apps: Simple, efficient and reliable? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 103–114.

Oliveira, W., Oliveira, R., & Castor, F. (2017). A study on the energy consumption of android app development approaches. In *Proceedings of the 14th International Conference on Mining Software Repositories*, 42–52.

Oliveira, W., Torres, W., Castor, F., & Ximenes, B. H. (2016). Native or web? a preliminary study on the energy consumption of android development models. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 1:589–593.

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

Pereira, R., Simão, P., Cunha, J., & Saraiva, J. a. (2018). jstanley: Placing a green thumb on java collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 856–859.

Peterson, M. (2009). *Decisions under ignorance*, 40–63. Cambridge Introductions to Philosophy. Cambridge University Press.

Pinto, G. & Castor, F. (2017). Energy efficiency: A new concern for application software developers. *Commun. ACM*, 60(12):68–75.

Pinto, G., Liu, K., Castor, F., & Liu, Y. D. (2016). A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 20–31.

Romano, J., Kromrey, J., Coraggio, J., & Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research*, 1–3.

S. Shapiro, S. & B. Wilk, M. (1965). An analysis of variance test for normality. *Biometrica*, 52:591–.

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*, 36:1–36:10.

Schmidt, A. ., Bye, R., Schmidt, H. ., Clausen, J., Kiraz, O., Yuksel, K. A., Camtepe, S. A., & Albayrak, S. (2009). Static analysis of executables for collaborative malware detection on android. In *2009 IEEE International Conference on Communications*, 1–5.

Wilks, D. S. (2011). *Statistical methods in the atmospheric sciences*. Elsevier Academic Press, Amsterdam; Boston.

Xu, G. (2013). Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, 1–26.

APPENDIX A – ENERGY PROFILE OF THE STUDIED COLLECTIONS

Figure 11: Non-thread-safe map operations for **note**

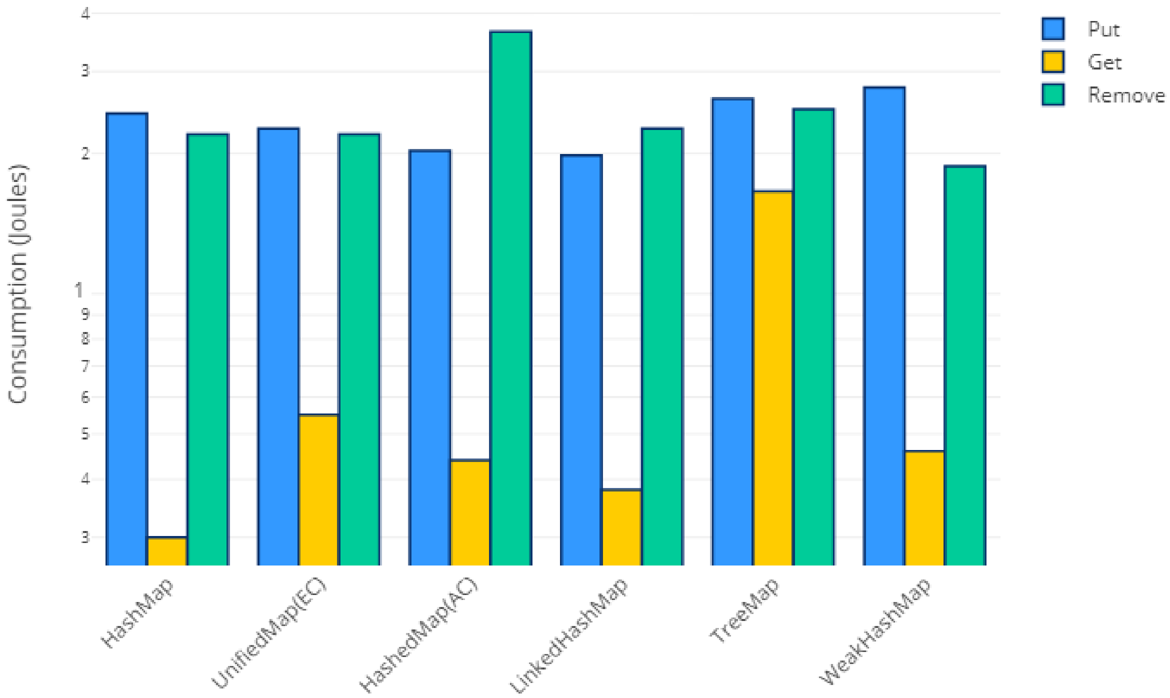


Figure 12: Non-thread-safe set operations for **note**

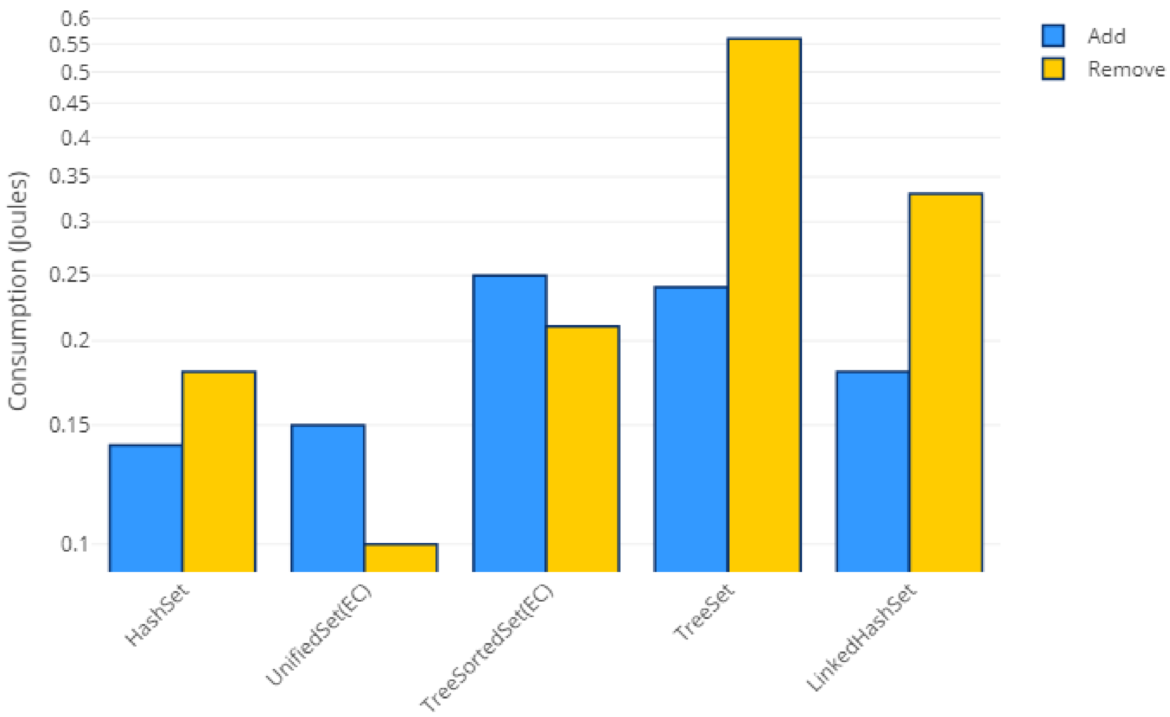


Figure 13: Thread-safe list additions for **note**

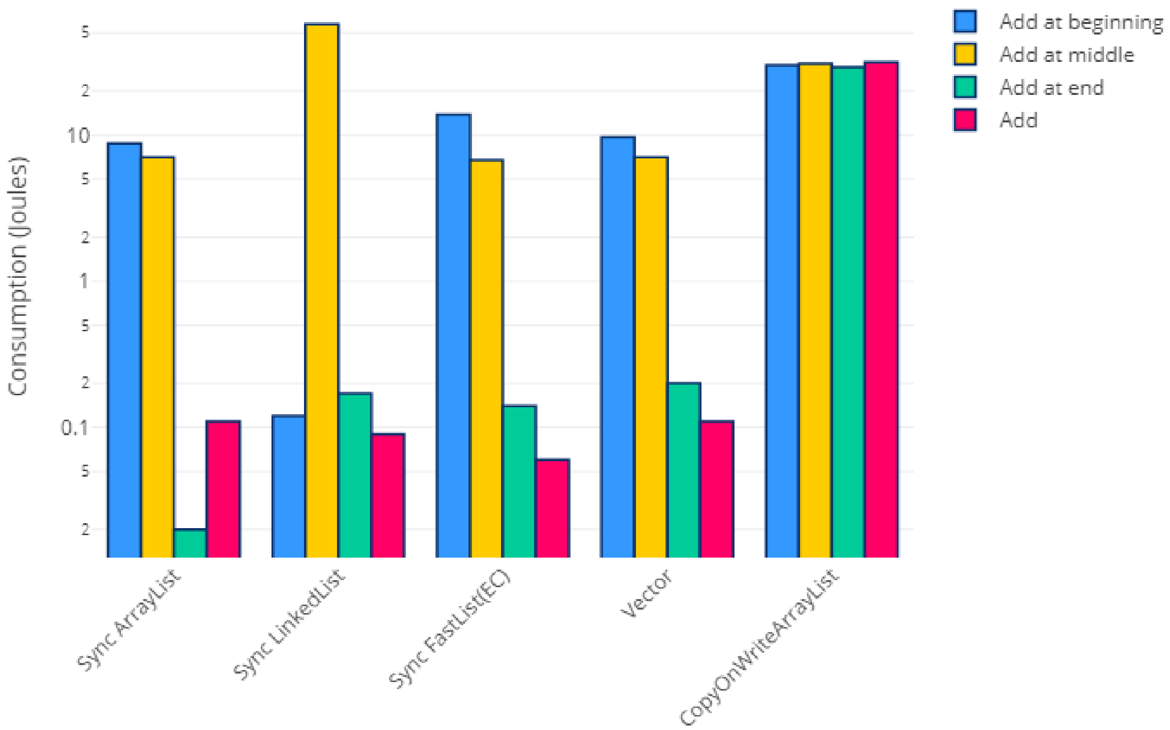


Figure 14: Thread-safe list removals for **note**

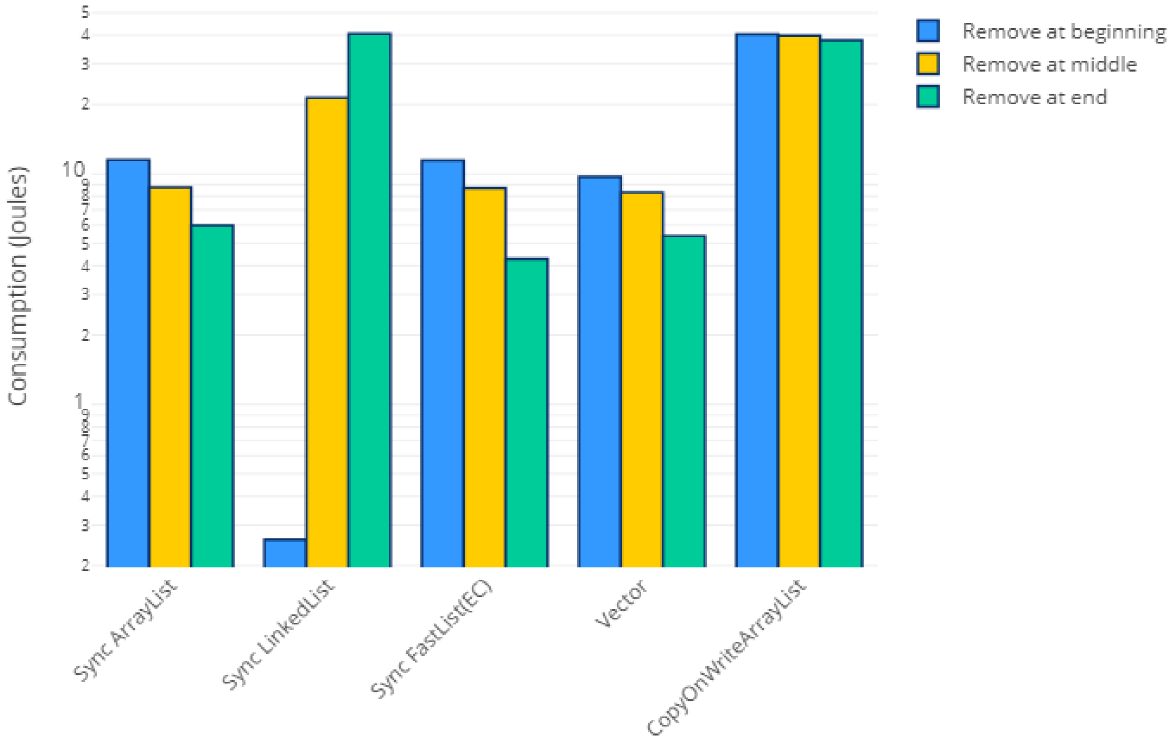


Figure 15: Thread-safe list traverse for **note**

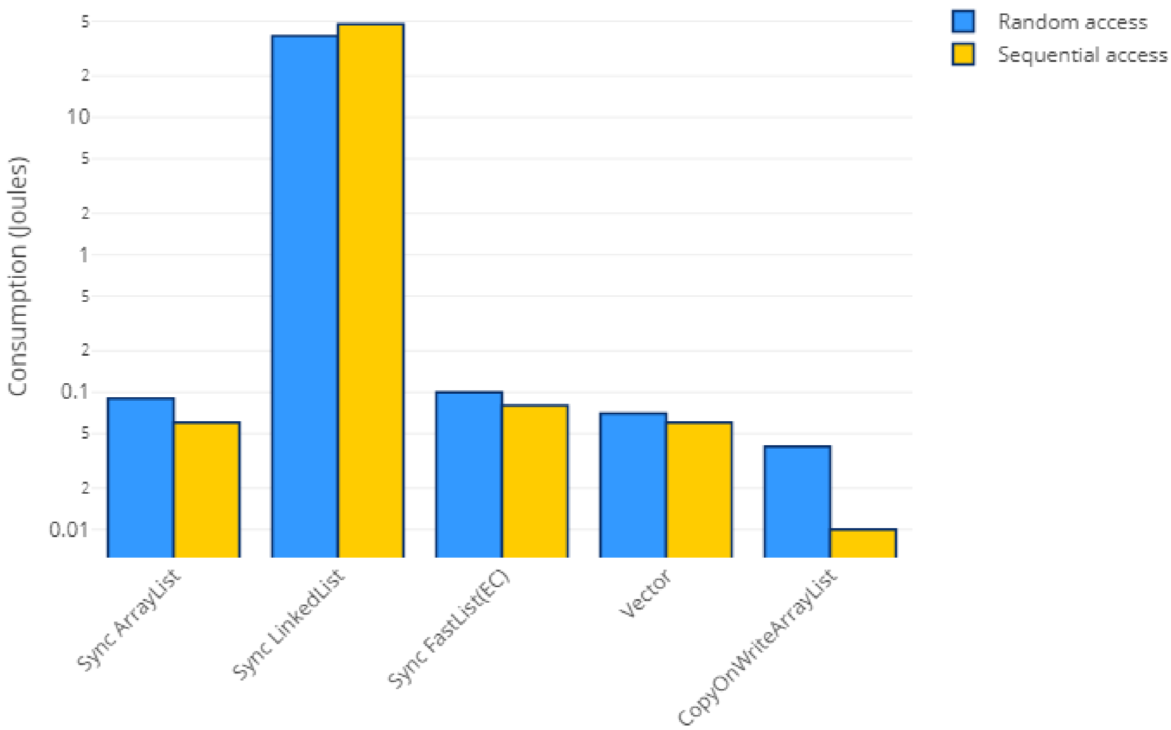


Figure 16: Thread-safe map operations for **note**