Comparing the Energy Consumption of Java I/O Libraries and Methods

Zakaria Ournani, Romain Rouvoy, Pierre Rust, Joel Penhoat

Evaluating The Energy Consumption of Java I/O APIs

Zakaria OURNANI
Orange Labs/ Inria / Univ. Lille
France
zakaria.ournani@inria.fr

Romain ROUVOY
Univ. Lille / Inria / IUF
France
romain.rouvoy@univ-lille.fr

Pierre RUST
Orange Labs
France
pierre.rust@orange.com

Joel PENHOAT
Orange Labs
France
joel.penhoat@orange.com

***Abstract*—The Java language is rich of native and third-party I/O APIs that most Java applications and software use. Such operations can even be considered core to most software as they allow the interaction with the user and its data in a non-volatile way. Yet, the I/O captivate a lot of attention due to their importance, but also due to the cost that these relatively slow operations add to read and write precious data, most commonly from/to disks. In this context, the impact of these I/O operations on energy consumption didn't get as much attention. Of course, I/O operations are responsible for energy consumption at the level of the storage medium (HDD or SSD) but they also induce non-negligible costs –both performance and energy-wise– at the CPU level. However, only few works take into account the impact of I/O on the energy consumption, especially at the CPU-level.**

Hence, this paper elaborates a detailed study with two main objectives. First we aim at assessing the energy consumption of several well-known I/O libraries methods, and investigate if different read/write methods can exhibit different energy consumption. Concretely, we assess —using micro-benchmarks— the energy consumption of 27 I/O methods for several file sizes and establish the truth about the most and least energy efficient methods. The second objective is to validate the results of the first experiments on real Java projects by substituting their default I/O methods and measuring the before/after energy consumption.

Our results showed that *i)* different I/O methods consume very different amounts of energy, such as NIO Channels that are 20% more efficient than other methods for read purposes *ii)* substituting the I/O method in a software by a more efficient one can save an important amount of energy, 15% of energy saving has been registered for K-nucleotide and 3% for Zip4j. We also showed that choosing the right I/O method can save more than 30% of energy consumption when using the Javax.crypto API.

Our work offers direct conclusions and guidelines on which I/O methods to use in which situation (read all data, read specific data, write data, etc.) for a better energy efficiency. It also opens doors for other works to better optimize the energy consumption of the I/O APIs and methods.

***Index Terms*—Energy consumption, Java, I/O.**

I. Introduction

Energy efficiency of software systems is undoubtedly a major challenge for the software engineering community. Beyond state-of-the-art key performance indicators, such as latency, throughput, scalability or availability, energy consumption challenges developers to reach the best performances while minimizing the subsequent resource requirements. In this context, previous studies in the field have been focusing on the impact of algorithms [1], [2] and data structures [3], [4] to reduce the energy consumption in presence of computation-intensive applications. Nevertheless, little effort has been invested in the study of the energy efficiency of Java I/O libraries [5] for the purpose of data-intensive systems/applications. Data-intensive applications are expected to process huge amounts of data for different purposes, from big data analytics to online cloud microservices. While the hardware components tend to keep improving on storage capacity and throughput, it remains unclear if their software counterparts succeed to keep the pace and provide energy-efficient solutions to efficiently read and write data. This is particularly difficult in Java, which provides a vast ecosystem of built-in functions and third-parties libraries to interact with persistent storage. In addition to this rich ecosystem, I/O APIs may be subject to the emergence of other features, like asynchronous capabilities (NIO), which can exhibit a different energy footprint. We believe that guiding the developers in picking the most energy efficient I/O method is a key challenge to deliver sustainable software with no compromise on the performances.

In this area, previous works have already shown the substantial contribution of I/O to the global energy consumption of software. For example, Lyu *et al.* [6] indicated that 10% of the mobile battery drains due to I/O operations. The energy consumption of Java I/O APIs has also been compared by Rocha *et al.* [5]. They reported that the energy consumption can widely vary among these APIs, and that the most popular APIs are not always the most energy efficient. This study delivers interesting insights about some APIs, such as `java.io.In(Out)putStream` and `java.io.Reader(Writer)`, as well as their inherited classes. However, it lacks some considerations for major I/O APIs, such as channels (`Java.nio.FileChannel`) or other popular third-party libraries (*e.g.*, Apache, Google Guava), it also did not consider different usage profiles of I/O, and failed to reproduce the experiments on a realistic Java project.

In this paper, we therefore assess the energy consumption of 27 different I/O methods issued from multiple native and third-party libraries using micro-benchmarks and different workloads. These methods are tested and compared for different scenarios and use cases (read the whole file, read a file part-by-part, seek data from a file, write data in a file, using a different buffer sizes, etc.). Concretely, the purpose of the study is to

answer the following research questions:

RQ 1: How do I/O methods affect the energy consumption of a Java code?

RQ 2: Can we reduce the energy consumption of software by substituting its I/O methods?

Our empirical exploration highlights the most energy-efficient methods for each use case. For instance, using channels for massive reads is usually 10–20% more energy efficient than other I/O methods. Moreover, we substituted the default I/O methods of well-known Java benchmarks and projects to effectively reduce their energy consumption.

Beyond answering these two research questions, the contributions of this paper can be summarized as:

1) Elucidate the energetic behavior of 27 different I/O methods issued from multiple libraries, using several file sizes,
2) Identify the most energy-efficient methods for several read and write use-cases,
3) Model the energy consumption in regards to the buffer size for the buffered I/O methods,
4) Deliver insights and guidelines to summarize the results and conclusion of our experiments,
5) Investigate the potential gain of substituting the default I/O methods of benchmarks and real java projects.

The remainder of this paper is organized as follows. Section II discusses the related works about energy consumption in Java software systems. Section III introduces the methodology (hardware, projects and experiments design) we adopted in this study. Section IV analyzes several experiments we conducted to evaluate the energy consumption of Java I/O methods, as well as the results we observed during these experiments. Finally, Sections V and VI cover the threats to validity and our conclusions, respectively.

II. Related Works

In this section, we review the state of the art related to Java energy efficiency. The energy consumption of Java applications and the JVM platforms gained interest recently. In particular, many studies have been conducted to reduce the energy consumption of Java applications from servers [7], [8], [9], [10], [11] to mobile devices [12], [13], [14], [15].

For example, the energy consumption of Java collections and data structures has been largely discussed by many papers [16], [17], [18], [10]. Pinto *et al.* [19] studied the effect of Java collections on energy consumption, including parameters, such as the collection size and the most executed operations (insertion, removal, search). Hasan *et al.* [3]compared the energy consumption of several Java data structures. They analyzed 6 real projects and measured the evolution of the energy consumption in different scenarios (insertion at the beginning, iteration, etc.). Their study showed that wrongly choosing the implementation behind a collection may degrade the energy consumption by up to 300%. Other works [20], [21] even delivered tools to enhance the energy consumption by replacing the used collections for several scenarios.

The energy consumption of Java code refactorings is another major issue that has been extensively studied [22], [23], [24], [25]. Sahin *et al.* studied the impact of 6 refactoring rules on a total of 197 code snippets found in 9 Java applications. Their results showed that the impact of applying the refactoring could be statically significant, but is not consistent across software and platform versions. They suggested that knowledge on the impact of refactoring rules on the energy consumption could be integrated within IDEs to help developers in building less energy-bleeding software. In a more detailed study of the impact of a single refactoring rule—known as "*inline method*"—on 3 Java applications, the authors [26] reported that the impact on the execution time and energy consumption that was expected to be positive, was not always true. In another paper, Pereira *et al.* [27] proposed SPELL, the energy leaks detector tool. The tool uses JRAPL [28] to measure energy consumption and detect energy inefficient code fragments using a statistical spectrum-based energy red spots localization. Some preliminary tests on Java programs resulted in up to 18% energy savings.

In this context, I/O have only been weakly addressed with a few studies focusing on the performance of Java I/O. For example, Karabyn *et al.* [29] deeply compared the performance and scalability of Java IO and Java NIO classes within servers. Their results showcased that—for long-lived connections—the difference in throughput is marginal, but that Java NIO has the edge when it comes to a lot of simultaneous connections and offers a minimal memory usage.

Pinto *et al.* discussed 12 contributions taken from the state of the art on the refactorings that could be applied to improve software energy efficiency [30]. This literature review was conducted on the papers that were published in 8 of the top software engineering conferences prior to 2015. It summarizes insights related to CPU offloading, HTTP requests, I/O operations, DVFS techniques, etc. For I/O operations, the paper reported that I/O utilities contribute significantly to the energy consumption of mobile applications.

For Java I/O libraries' energy consumption, Rocha *et al.* [5] conducted a comparative study of some I/O methods, mostly I/O classes that inherit from `java.io.In(Out)putStream` and `java.io.Reader(Writer)`. This preliminary study offers some interesting insights on the behavior of some of the most common native Java I/O methods. Nevertheless, the reported work lacks some of the most used I/O methods, and does not deliver guidelines to reduce the energy consumption across multiple I/O scenarios and use cases.

The objective of our work is to establish a robust assessment of the energy consumption of some mainstream native and third-party I/O libraries. Concretely, we aim at understanding the energetic impact of some I/O libraries that have not been covered, such as NIO channels, or third-party libraries, such as Apache or Guava. Moreover, we investigate multiple scenarios and use cases, such as loading the whole file in memory, reading a file by chunks, seeking specific data from a file, and writing data. We also cover the energy consumption of using different buffer sizes for buffered operations. Finally, we assess the most energy-efficient methods on acknowledged

benchmarks and public Java projects to reduce the overall energy consumption by substituting default I/O methods with the ones that reported a better energy efficiency.

III. Methodology

To investigate the energy impact of different Java I/O libraries and methods, we conducted a wide set of experiments using several micro-benchmarks.

A. Environment Settings

To report on reproducible measurements, we used the same hardware configuration for all our experiments. The machine is equipped with Intel i7-6600U @ 2.60GHz, 16 GB of RAM and an Intel SSD Pro 5450s Series, with a capacity of 256 GB (up to 550 MB/s for reading and 500 MB/s for writing operations). Our experimental protocol enforces that the software under test is the only process executed on the node, configured with a very minimal version of Ubuntu 18.04 (4.15.0-140-generic kernel version) with OpenJDK 1.8.0_242. The minimal configuration of the OS indicates an operating system with only the required daemons and services. Everything else was disabled [31], to minimize the variations on the observed energy consumption. We focused on measuring the CPU energy consumption to evaluate the differences in energy consumption between the tested methods. We thus used Intel RAPL as a physical power meter to analyze the energy consumption of the CPU. RAPL is one of the most accurate tools to report on the global energy consumption of a processor [32], [33]. The registered CPU energy consumption includes core and uncore energy consumptions.

Moreover, we tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [34]. Every single experiment, therefore, reports on energy metrics obtained from 30 executions per benchmark. All of our experiment materials are made available for use/reproducibility from https://anonymous.4open.science/r/7f5f13601f85a2446a9f934.

B. Experiments Design

Our experiments are structured in two steps. The first step has an exploratory nature: by testing and comparing numerous Java I/O libraries and methods to read and write data from files. To do so, we prepared text and binary files of several sizes. Table I summaries the size on disk of each file type and the lines count $\times$ lines length for the equivalent text file of the same size. Next, we create micro-benchmarks for each I/O method, then run them for 30 times and measure the execution time and the CPU energy consumption. All read benchmarks run the same function `consume` to compute a hash from the raw input data, thus prevent the JIT from discarding some source code and altering the expected behavior.

The second step aims at validating the results of the first step using some Java benchmarks from the *Computer Language Benchmarks Game* (CLBG): Fasta and K-nucleotide, but also using a real Java project (Zip4J) and a real Java API (Javax.Crypto). This validation is achieved by substituting the read/write methods used in these benchmarks/projects by the ones that exhibited the lowest energy consumption in the previous step, and check whether the energy consumption of these source codes can be reduced.

TABLE I: The list of file sizes.

Method	File Size	Lines count	Line length
Tiny	100 KB	1,000	100
Small	15 MB	100,000	150
Medium	200 MB	1,000,000	200
Medium-large	3.2 GB	8,000,000	400
Large	16 GB	40,000,000	400

1) Java I/O Libraries: We evaluated a wide set of read/write methods from several Java I/O libraries in our study. The full list of methods is provided in Table II. The purpose is not to explore every sub-classes or sub-methods of the ones present in Table II. For example, we do not test every method that extends `InputStream`, such as `PushbackInputStream` or `ByteArrayInputStream`. Such methods have already been compared in [5]. The purpose is rather to study and compare different methods issued from different classes, libraries, and even famous third-party solutions. Table II reports on 3 different purposes: "Read" to read data, "Seek" for accessing some data at a specific position and "Write" to write data. "ReadAll" is a particular case of Read where we read the content of the whole file at once. Most of the used classes are issued from `java.io` and `java.nio` (NIO). NIO is for non-blocking I/O—*i.e.*, by handling the I/O operations in a non-blocking way using buffers, channels, etc.

Some of the methods in Table II can use a buffer, which is a memory block of a given size (usually 8,192 bytes). Contrary to arrays or lists, a buffer has a limit that differs from its capacity. This enforces the ability to have a variable size up to the capacity, which is the maximum it can handle. Moreover, the buffer offers a built-in way to read or write the next element, easing sequential processing, and a way to save the current position for later reset (mark and position functions, respectively). We note that the versions of the used Apache and Guava libraries are 2.8.0 and 23.0 respectively.

2) Real Benchmarks: The impact of I/O operations on software energy consumption mainly depends on the type of software. This impact could be high for some applications, such as file compression or serialization applications, while it can be much lower for other classes of applications that perform much less I/O operations. Hence, we choose benchmarks that stress the usage of I/O operations to confirm the results of the micro-benchmarks and clearly spot the differences in energy consumption that such operations can cause for real applications.

First, we choose 2 I/O focused benchmarks from the *Computer Language Benchmarks Game*: Fasta, and K-nucleotide. The first one generates DNA sequences (CGAT), by weighted random selection from 2 alphabets and outputs the results in a file. The second benchmark reads line-by-line the Fasta

TABLE II: The list of the studied I/O classes and methods

Class	Acronym	Method	Purpose	Availability	Description
java.io.InputStreamReader	IOSTREAM	read(Byte[])	Read	JDK 1.0	InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset
java.io.OutputStreamWriter	IOSTREAM	write(String)	Write	JDK 1.0	OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset
java.io.FileInputStream	IOSTREAM	Skip(long) readAllBytes()	Seek ReadAll	JDK 1.0	A FileInputStream obtains input bytes from a file in a file system. It is meant for reading streams of raw bytes such as image data
java.io.BufferedInputStream	BIOSTREAM	readLine()	Read	JDK 1.0	A BufferedInputStream adds the ability to buffer the input and to support the limit, in addition to the mark and reset methods
java.io.BufferedOutpuStream	BIOSTREAM	write(String)	Write	JDK 1.0	It Implements a buffered output stream. It can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written
java.io.FileReader	FILEREADER	read(char[])	Read	JDK 1.1	FileReader is meant for reading streams of characters
java.io.FileWriter	FILEWRITER	write(String,int,int)	Write	JDK 1.1	FileWriter is meant for writing streams of characters
java.io.BufferedReader	BFILEREADER	readLine()	Read	JDK 1.1	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines
java.io.BufferedWriter	BFILEWRITER	write(String)	Write	JDK 1.1	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings
java.nio.channels.FileChannel	CHANNEL	read(ByteBuffer) write(ByteBuffer) position(long)	Read Write Seek	JDK 1.4	A Filechannel is a SeekableByteChannel that is connected to a file. It has a current position within its file which can be both queried and modified
java.nio.FileChannel	OMCHANNEL	map(MapMode,long,long)	Read	JDK 1.4	Maps a region of this channel's file directly into memory
java.nio.Files	NIOF	readLine() write(String) readAllLines(Path)	Read Write ReadAll	JDK 1.7	This class consists exclusively of static methods that operate on files, directories, or other types of files. It delegates to the associated file system provider to perform the file operations
java.util.Scanner	SCANNER	nextLine()	Read	JDK 1.5	A simple text scanner which can parse primitive types and strings using regular expressions
java.io.RandomAccesFile	RAF	readLine() writeBytes(String) seek(long)	Read Write Seek	JDK 1.0	A random access file behaves like a large array of bytes stored in the file system with a pointer into the implied array, used for both read and write operations
apache.commons.io.FileUtils	APACHE	readFileToString(File,Charset) write(File,List¡String¿,Boolean) readFileToByteArray(File, Charset)	Read Write ReadAll	NA	Apache General file manipulation utilities. It offers reading, writing, and much more operations
google.common.io.CharSource google.common.io.CharSink google.common.io.Files	GUAVA	readLine() write(String) readLines(file, Charset)	Read Write ReadAll	NA	Google library that provides utility methods for working with files, including reading and writing operations.

format of the previous file, extracts the DNA sequence number Three, and updates a hash-table of k-nucleotide keys to count values within a particular reading-frame.

We substitute the write and read methods from the Fasta and K-nucleotide with the methods that exhibited the best energy efficiency from our micro-benchmarks. The purpose is to check if we can reduce the energetic impact of those benchmarks with only the substitution of the I/O method.

Moreover, we apply the same substitution process on the read method of real Java projects: Zip4J[1] and Javax.Crypto.[2] The first project is a Java library that offers many operations for handling zips and streams. It is the only Java library with support for zip encryption, apart from several other features. The second project is a Java API that delivers many classes and interfaces for cryptographic operations. Substituting the I/O operations in this context aims to deliver a more realistic feedback on the energy impact of I/O on a library that uses a fair amount of I/O operations, such as Zip4J and Crypto.

[1]https://github.com/srikanth-lingala/zip4j

[2]https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

IV. Experiments & Results

In this section, we expose the results of our experiments in order to answer our research questions.

A. Behavior of I/O Methods

In this part, we study the behavior of the considered I/O methods used in the micro-benchmarks. One benchmark has been written for each method of Table II. Then, experiments are run with each benchmark and for several file sizes (cf Table I).

1) Reading the Whole File at Once: This method is limited by the memory size—*i.e.*, it can only be applied on small files where the RAM size would allow loading the content of the whole file at once. Some of the classes mentioned in Table II natively offer the possibility to read the whole content of a file (ReadAll methods from IOStream, Apache, NIOF, Guava). While we believe that this way to read files is not compatible with all needs and thus is not the most interesting, we still run a quick comparison of the available methods using the medium file at best due to the memory limitation. Figure 1 depicts the energy consumption of 5 native methods to read the whole content of binary and text (strings or lines) files issued from 4 classes. It shows that the `InputStream.readAllBytes()` method (IOStream) is the most energy efficient, followed by Apache `ReadFileToByteArray(File, Charset)` or `ReadFileToString(File)`. NIO `Files.readAllLines(Path)` and Guava `readLines(File, Charset)` methods consumed the most energy among the 5 tested methods.

Despiste reading being limited by the file sizes, we can still notice a substantial gain in energy consumption with the appropriate method and use case. In fact, IOStream consumed 4 times less energy compared to Guava and 3 times less compared to NIOF, for a medium binary file. It was the most efficient way to read whole files among the 5 methods and is 60% more efficient than Apache to read binary files for medium file sizes. To natively read text files, Apache* was the most efficient way to do it, 40% more efficient than NIOF and 100% more efficient than Guava.

2) Reading the Whole File by Chunks: This way of reading is much more flexible and bypasses the memory size limitation by reading the file by chunks. Usually, this is achieved by using a buffer or an array of a given size that slides through the file to read input data. Many libraries offer functions to natively achieve this. We thus run read methods from Table II with different file sizes and assess their energy consumption. Figure 2 overviews the energy consumption of all tested read methods on several file sizes. The observed difference in energy consumption seems to be very small for our tiny, small, and medium file sizes, but the lines become clearly distinguishable as the file sizes grow. We note that two other read methods (RAF and Scanner) have been tested, but are not depicted in the figure for visualization issues. Our experiments in Table III showed that different input methods consume different energy. The most extravagant case is RandomAccessFile where we

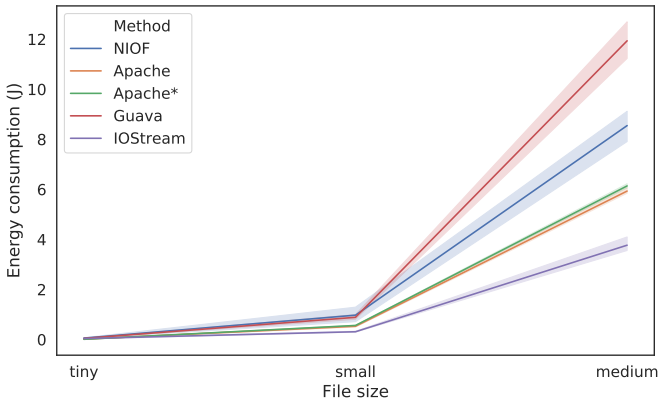


Fig. 1: Energy consumption to read the whole content of files.

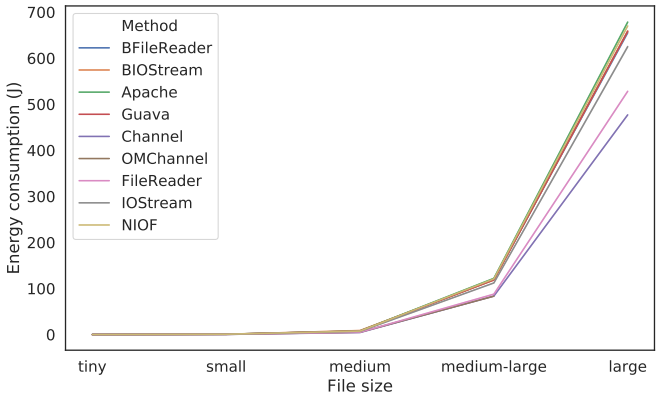


Fig. 2: Energy consumption to read files by chunks for several file sizes.

noticed an extra energy consumption, even for tiny and small files. This extra energy becomes very important for bigger files (up to 200 times more energy). The Scanner read is the second noticeable method that gave much worse results, compared to the others for all file sizes (more than 4 times more energy consumed with Scanner).

The other results of Table III are much closer, with methods that give very similar results for all file sizes (Apache, BFileReader, BIOStream, Guava, NIOF). The clear winner here are Channel and OnMemoryChannel that consumed the least energy among all methods and across all file sizes.[3] Along our experiments, NIO Channels consumed up to 20% less energy, compared to the average and about 10% to the second-best method (FileReader), which constitute a substantial gain, especially for large files and applications that use a good amount of I/O.

Finally, we noticed that using the buffer for FileReader and InputStream is not very beneficial. The buffered BFileReader and BIOStream consumed up to 10% more energy for large files.

[3] OnMemoryChannel could not be used on a large file due to memory limitation, as it uses memory mapping.

TABLE III: Energy consumption (joules) and execution time (ms) for reading files of different sizes by chunks.

Method	Tiny		Small		Medium		Medium-Large		Large	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Apache	0.03	1.9	0.68	58	7.7	724	122	$11.5 * 10^3$	678	$62 * 10^3$
BFileReader	0.01	1	0.69	61	7.7	714	119	$11.3 * 10^3$	655	$60 * 10^3$
BIOStream	0.01	1	0.7	60	8.6	748	118	$11.2 * 10^3$	658	$60 * 10^3$
Channel	**0.02**	1.5	**0.32**	27	**4.4**	434	**83**	$7.9 * 10^3$	**476**	$43 * 10^3$
FileReader	0.01	1	0.43	35	5	500	90	$9 * 10^3$	568	$50 * 10^3$
Guava	0.03	1.8	0.67	57	7.4	707	118	$11.2 * 10^3$	658	$60 * 10^3$
IOStream	0.03	1	0.49	44	7	683	112	$10.9 * 10^3$	625	$58 * 10^3$
NIOF	0.01	1.1	0.7	58	7.5	716	120	$11.5 * 10^3$	670	$62 * 10^3$
OMChannel	**0.01**	1	**0.3**	25	**5.1**	488	**83**	$7.9 * 10^3$	NA	NA
RAF	1.04	86	120	1200	1528	157062	24548	$25 * 10^5$	$1.2 * 10^5$	$13 * 10^6$
Scanner	0.1	8	3.1	230	19.4	1697	563	$5 * 10^4$	2893	$26 * 10^4$

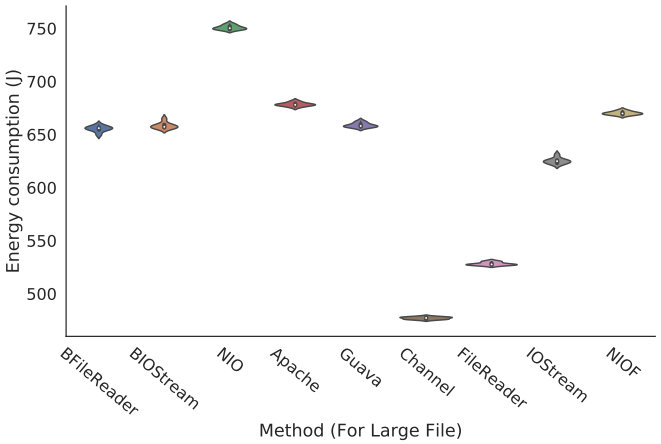


Fig. 3: Energy consumption of read methods for a large file.

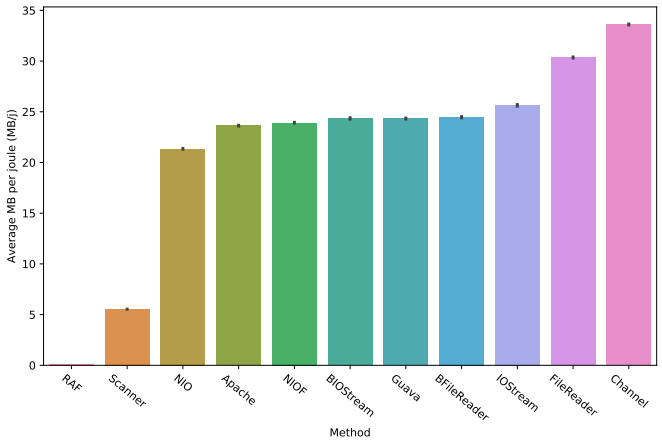


Fig. 4: The average of read data per joule for each method.

Figure 3 exposes more visually the results of Table III for a large file. The violin plots show very stable energy consumption values with very small standard deviations. This reports on robust experiments, but also validates the results of Table III, as all results are tightly centered around the average/median. The figure also allows to establish an easier comparison of the read methods issued from the different classes (RAF and SCANNER have been excluded for a better visualization, OMCHANNEL is not applicable for large files).

Figure 4 depicts the average of read data per joule for each method, while reading the large file. This confirms that the CHANNEL is more efficient and reads more data per joule. In fact, the CHANNEL read method reads about 35 MB/j. That is 5 MB/j more efficient than FILEREADER and 10 MB/j better than the average. RAF and SCANNER on the other hand are the least efficient with 5 MB/j and less than 1 MB/j, respectively.

One more question we wanted to address is: what is the most energy-efficient way to read data from a file if the memory size and the JVM heap size are sufficient to load the whole file? If we compare the necessary energy consumption to read our medium file in Figure 1 and Table III, we notice that reading the whole file at once with IOSTREAM consumed about 3.8 joules. That is at least 13% less consumed energy than any other method in Table III (4.4 joules being the lowest). This indicates that reading the whole file can be more energy efficient than reading it by chunks, when possible.

a) Buffer Size: Many of the previous methods require or use a buffer to perform the reading operation (BIOSTREAM, BFILEREADER, CHANNEL, OMCHANNEL, RAF, etc.). This buffer size may vary and can be set by the developer. In this experiment, we check the effect of the buffer size on the energy consumption.

Thus, we run multiple executions of the BIOSTREAM read operation (`BufferedInputStream`), and we vary the buffer size from its default value used for the previous experiments ($8,192$ bytes).

This difference is not noticeable for small files, but can be clearly seen in Figure 5 for a large file. This figure illustrates

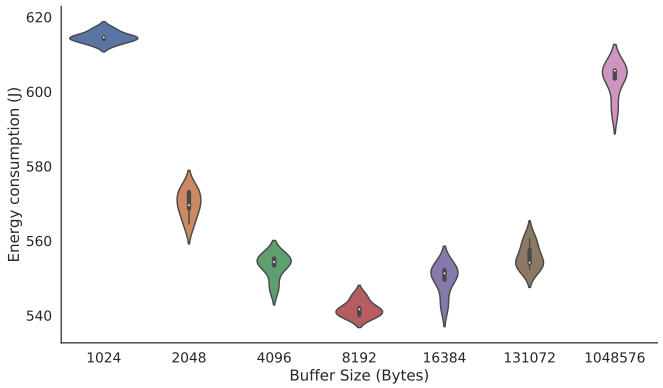


Fig. 5: Energy consumption to read files with different buffer sizes.

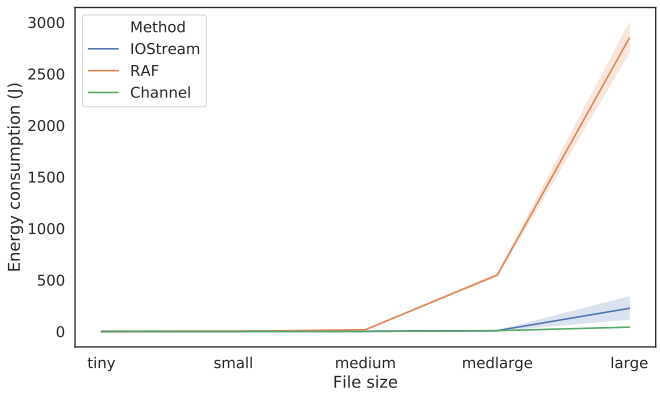


Fig. 6: Energy consumption to seek data from files of different sizes.

the measured energy consumption for each buffer size. We notice a parabolic shape of the values of energy consumption, approximately centered around 8192 bytes. The difference in energy consumption is very small (less than 3%) going to the nearest values to $8,192$ bytes ($4,096$ bytes and $16,384$ bytes). However, this difference grows much higher for further small or big values of the buffer size. Here, up to 13–15% more energy consumption for the buffer sizes $1,048,576$ bytes and $1,024$ bytes, respectively. Hence, the buffer size should not be too small or too big for a better energy efficiency.

3) Seeking Specific Data From a File: This represents the third possibility to read data from a file. It consists of moving a cursor or a pointer within a binary file, and read an amount of data. This way is very interesting to access some of the data within the file without reading the whole file, especially for very large files. One potential application relates to database files that contain an index, such as SQLite, where we need to access the data pointed by the index without reading the whole database.

Thus, we ran multiple experiments to compare the available seek methods for Table II and their energy consumption. To do so, we access and read different data at numerous positions within the files of multiple sizes. Each time, we read 100 bytes and move forward with $10,000$ bytes to read the next 100 bytes, until the end of the file.

Table IV and Figure 6 show the difference in energy consumption between the 3 different methods issued from CHANNELS, IOSTREAM, and RAF. Here again, the results show that using the NIO channels is the most energy-efficient way to seek data from files. This efficiency is mostly noticeable for a large file with a high number of seek operations, where CHANNELS position method is 5 times faster than IOSTREAM skip method. On the other hand, the RAF seek method is the slowest and the most energy consuming. It consumed 67 times more energy than NIO Channels for this experiment on a large file.

4) Writing a File: The write operation consists of physically writing data on a file in its disk location. Like reading, there are plenty of classes that offer write methods to achieve that purpose. Therefore, we assess the energy consumption of the write methods from Table II and try to identify the "good" and "bad" methods regarding the energy efficiency with different file sizes. Writing a file of a specific size is achieved by generating random text data of the appropriate size (LinesCount $\times$ LineLength) as described in Table I and writing it into a file.

Figure 7 depicts the evolution of the consumed energy by the different methods with growing file sizes (GUAVA has been omitted for a better visualization), while Table V gives the detailed energy consumption and the execution time of each method, and for each file size.

The results highlight that 4 of the tested methods are way more energy-consuming than the others. First, GUAVA's write method registered the worst energy efficiency, compared to all other methods and, for all file sizes, up to 6 times more energy consumption compared to the best-tested methods. Second, the NIOF writing operation consumed 50–100% more energy across all file sizes. Third, the NIO channels that gave the absolute best results for file reading are less efficient for the writing operation with 30% more energy consumption for large files. RAF is in the fourth position of the methods that performed worse than the others for write operations. It consumed 15% more energy compared to the best method.

The remaining methods (APACHE, BFILEWRITER, BIOSTREAM, FILEWRITER, IOSTREAM) gave very similar results, and were the most energy-efficient methods to write data into files.

Figure 8 delivers a better comparison of the energy consumption of the write methods for a large file.

One thing we noticed by comparing Figure 3 and Figure 8 is that writing a file consumes more energy than reading a file of the same size. In fact, we needed less than 500 joules to read our large file with the most efficient read method, while the least-consuming write method needs 5 times that amount of energy (2500 joules) to write a file of the same size. This can be clearly seen through Figure 9 that shows the average written data per joule for each method. The maximum value we observe for the most energy efficient write operations is

TABLE IV: Energy consumption (joules) and execution time (ms) for seeking data from different files.

Method	Tiny		Small		Medium		Medium-Large		Large	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Channel	0.01	1.2	0.07	7.6	0.5	49.5	7.7	766	**42**	$4.1 * 10^3$
IOStream	0.01	1.1	0.06	7.6	0.6	62	8.3	814	225	$8.2 * 10^3$
RAF	0.01	1.6	1.2	111	17	1693	547	$5.3 * 10^4$	2847	$2.7 * 10^5$

TABLE V: Energy consumption (joules) and execution time (ms) for writing files of different sizes by chunks.

Method	Tiny		Small		Medium		Medium-Large		Large	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Apache	0.20	13.3	8.4	760	8.7	$7.8 * 10^3$	679	$6.2 * 10^4$	2526	$2.3 * 10^5$
BFileWriter	0.24	15.1	9.5	794	8.6	$7.8 * 10^3$	$6.1 * 10^4$	$11.3 * 10^3$	2501	$2.3 * 10^5$
BIOStream	0.22	13.7	8.9	770	82	$7.5 * 10^3$	662	$6.0 * 10^4$	2502	$2.3 * 10^5$
Channel	0.23	14.5	10	919	99	$9.1 * 10^3$	798	$7.3 * 10^4$	3293	$2.8 * 10^5$
FileWriter	0.15	10.5	8.4	764	83	$7.6 * 10^3$	669	$6.1 * 10^4$	2518	$2.3 * 10^5$
Guava	1.18	80.7	95	6882	962	$7.1 * 10^4$	7507	$5.5 * 10^5$	15592	$1.7 * 10^6$
IOStream	0.12	9	8.5	765	83	$7.5 * 10^3$	671	$6.1 * 10^4$	2522	$2.3 * 10^5$
NIOF	0.32	22.3	27	1714	238	$1.5 * 10^4$	1897	$1.2 * 10^5$	3684	$3.4 * 10^5$
RAF	0.14	10.6	10	920	98	$9 * 10^3$	795	$7.3 * 10^4$	2932	$2.6 * 10^5$

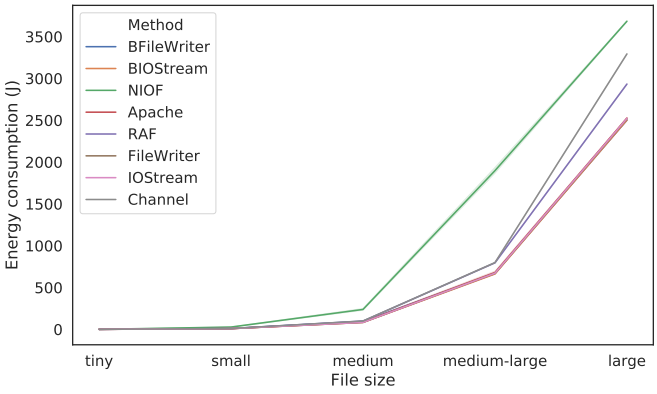


Fig. 7: Energy consumption to write files for several file sizes.

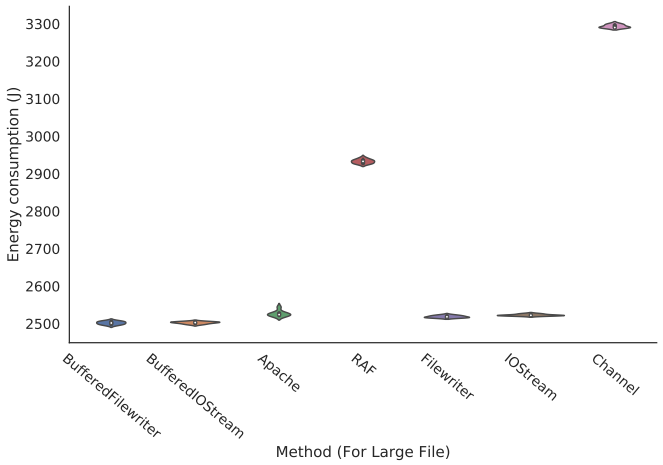


Fig. 8: Energy consumption of write methods for a large file.

7 MB/j. This is much lower than the values obtained for read operations (up to 35 MB/j in Figure 4).

To answer RQ1, we conclude that using different I/O methods can alter the software energy consumption. Our experiments delivered some insights and guidelines that can be summarized as:

1) For read methods, using the class `nio.FileChannel` proved to be the most appropriate choice to consume the least amount of energy while reading files of different sizes. It was at least 10–20% more energy efficient;
2) SCANNER and RAF reported on a very high energy consumption, compared to the other methods and should thus be carefully used, if not avoided, for data reading purposes;
3) For methods that use buffers, using buffer sizes that are too big or too small compared to the default size of $8,192$ bytes may introduce an extra cost in energy consumption;
4) Reading the whole file at once is limited by the file/memory size, but can be very energy efficient;
5) For write operations, many methods reported on a similar energy consumption, but other methods, such

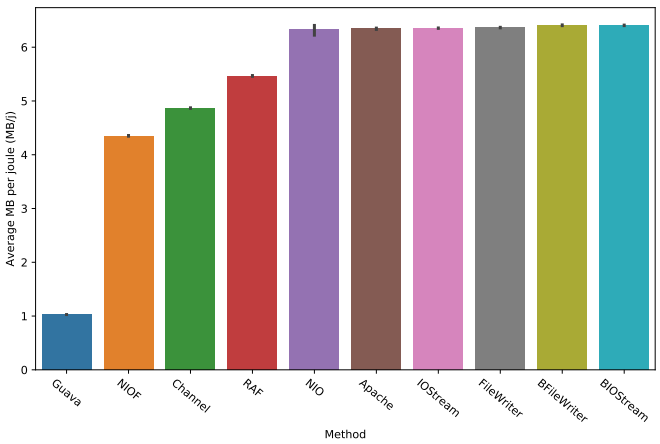


Fig. 9: The average of written data per joule for each method.

> as GUAVA or NIOF consumed more energy, and should thus be avoided.

B. Substituting I/O Methods

In this part, we aim at answering the second research question by substituting and comparing multiple I/O methods on 4 Java projects presented in Figure 10: Fasta, K-nucleotide, Zip4J, and Javax.Crypto. Thus, we substitute the write method from the Fasta program with the methods that reported on a low energy consumption from the previous step to save a 1.5 GB of generated nucleotides. We then assess the energy consumption with each of the methods used, for 30 executions each.

Figure 10a illustrates the boxplots that summarize the energy consumption of each method's execution. As seen in the previous section, the write methods are almost similar and consume approximately the same amount of energy (around 255 joules). The written Fasta data is used as an input file for the K-nucleotide program. Figure 10b depicts the energy consumption of the nucleotide read operations using the methods that we used as substitutes to the default BIOSTREAM read method (the methods that performed best on the micro-benchmarks). Just like in our previous micro-benchmark experiments, the figure shows that using CHANNELS results in a lower energy consumption (15% less energy consumption using CHANNELS compared to the default BIOSTREAM).

This result can be confirmed by Figure 10c in which we substituted the reading method of the Java zipping library with other read methods susceptible of having an equivalent or better energy efficiency. Then, we ran experiments where we zipped our large file and saved the result. Here again, using CHANNELS causes a reduction of 3% in energy consumption, compared to the default INPUTSTREAM method. The gain of using channels is only 3% in this experiment because the file reading phase only covers 10–15% of the total execution time of the zipping process. Finally, Figure 10d represents another confirmation of the energy efficiency of using CHANNELS for file reading purposes. In this example, using CHANNELS was at least 30% more energy efficient than FILEREADER and IOSTREAM (50% more energy efficient than BIOSTREAM and NIOF) to decrypt a 3 GB file using the Crypto API with the AES algorithm and a 16 bytes key.

> To answer RQ2, we empirically showed that we can reduce the energy consumption of software and programs that run a substantial amount of I/O by choosing the right methods. Using NIO Channels proved to be very energy efficient here again on K-nucleotide, Zip4J, and Crypto, compared to other read methods.

V. THREATS TO VALIDITY

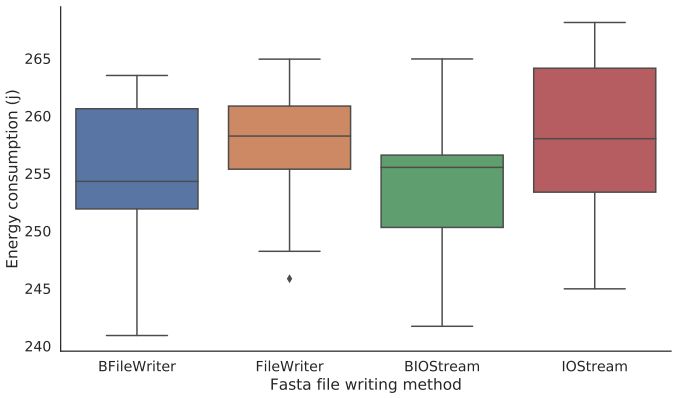
There are a couple of issues that might impact the accuracy of our results. Starting with the use of the Intel RAPL to measure the energy consumption. Although RAPL is one of the most accurate available tools to measure the energy consumption of software [32], [33], it only gives the global energy consumption and no fine-grained measures at process or thread levels. Hence, we used bare-metal hardware with a minimal OS configuration. We disabled all the non-essential services and daemons to limit the overhead that the OS may add to the execution [31]. We also ran all of our experiments within the exact same node to avoid the problem of CPU energy consumption variation [35], [31].

The execution time and registered energy consumption for short tasks is one more subtle threat to validity. In fact, some read and write experiments on tiny and small files are very fast, so we cannot assess the energy consumption faithfully. To overcome this issue, we constituted every execution of these fast experiments of many iterations. Most importantly, we focused all of our results analysis and conclusions on the experiments that last much longer, using our medium and large files. Java *Just-in-time* (JIT) compiler might constitute another threat to the validity of this work, especially for read operations in micro-benchmarks, if the read data is not used. Thus, we executed a hash method that consumes the read data similarly to a black-hole in JMH (*Java Microbenchmark Harness*). We also discarded the whole JVM instance between executions, so the JIT does not cache data between executions and alter the measures. We did not disable the JIT because the study would not reflect a real usage of I/O methods in realistic Java applications anymore.

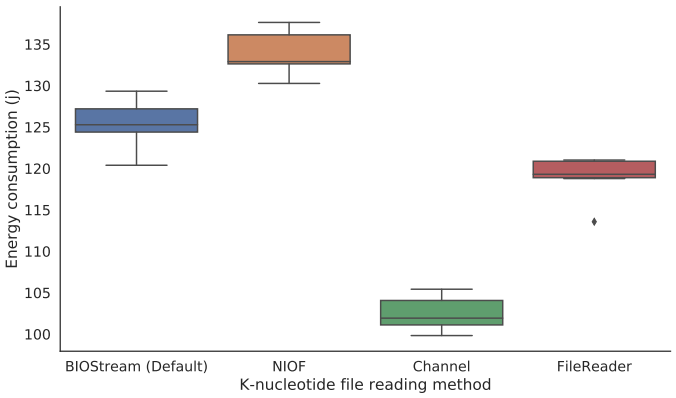
The considered I/O libraries come with multiple read and write operations. To conduct this first study, we were obliged to select some of these methods. Our aim was to diversify our methods selection to deal with both binary and text files, but we also wanted to select methods with similar signatures, so we can construct a fair and relevant comparison between the I/O libraries. We plan on generalizing our study to a larger set of methods, with a much deeper validation phase that would include dozens of Github Java projects, for which we can submit pull requests to reduce their energy consumption by substituting the I/O methods.

VI. CONCLUSION

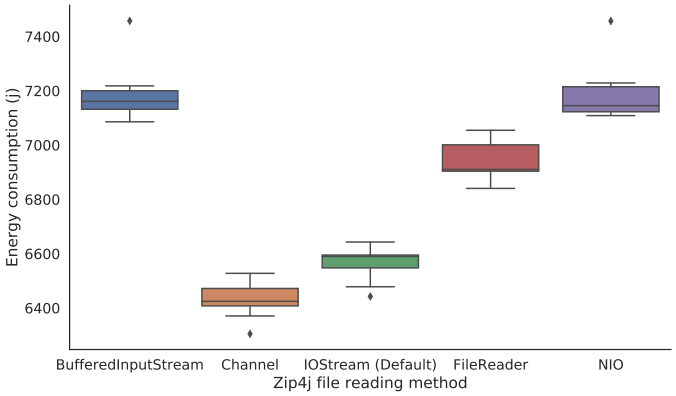
This paper reports on an empirical investigation of the key differences in energy consumption of some famous Java I/O



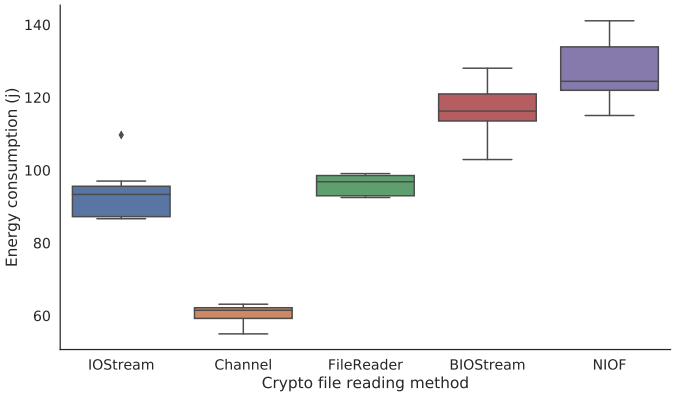
(a) Energy consumption of Fasta to generate and write 1.5 GB of nucleotide.



(b) Energy consumption of K-nucleotide to to read the data generated by Fasta.



(c) Energy consumption of Zip4J to zip our large file with different read methods.



(d) Energy consumption of the Crypto library to decrypt a 3 GB file with different read methods.

Fig. 10: Comparing the energy consumption of Fasta with multiple write methods, K-nucleotide with multiple method to read the Fasta generated file, Zip4j to read and zip our large file using different read methods, and the Crypto library to decrypt a 3 GB file.

libraries and their read and write methods. Concretely, we assessed the energy consumption of 27 different methods using dedicated micro-benchmarks regarding several scenarios: read the whole file at once, read the file by chunks (with optimal buffer size), seek specific data within a file and write data to a file. Our experiments showed that not all read and write methods exhibit the same energy consumption while reading or writing data. On one hand, some methods can be very efficient, such as using NIO channels for read operations. On the other hand, other methods can be very inefficient, such as using *Random Access File* to read or write data.

To validate our results, we substituted/compared I/O methods on 4 real benchmarks and real Java projects with methods that registered a good energy efficiency with micro-benchmarks. We were able to reduce the energy consumption on three of them by using NIO Channels, achieving 15%, 3%, and 30% energy savings for K-nucleotide, Zip4j and Javax.Crypto, respectively.

For future works, we will consider a massive reproduction of our results on a large set of projects. We plan on substituting the default I/O methods on multiple Github Java projects, assess the energy consumption, and open pull requests on those projects to enhance their energy efficiency. We also aim at automating a plug-in that detects energy-consuming I/O methods and recommends more energy-efficient alternatives.

REFERENCES

[1] E. Jagroep, G. Procaccianti, J. M. van der Werf, S. Brinkkemper, L. Blom, and R. van Vliet, "Energy efficiency on the product roadmap: An empirical study across releases of a software product: Energy efficiency on the product roadmap," *Journal of Software: Evolution and Process*, vol. 29, no. 2, p. e1852, Feb. 2017.

[2] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 237–248.

[3] Samir Hasan, Rachary King, and Munawar Hafiz, "Energy Profiles of Java Collections Classes," in *ICSE*, 2016.

[4] L. Cruz, R. Abreu, and J. Rouvignac, "Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 205–206.

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

[6] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An empirical study of local database usage in android applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 444–455.

[7] M. Longo, A. Rodriguez, C. Mateos, and A. Zunino, "Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings," *Computer Science and Information Systems*, vol. 16, no. 2, pp. 541–564, 2019.

[8] S. Lafond and J. Lilius, "An Energy Consumption Model for an Embedded Java Virtual Machine," in *Architecture of Computing Systems - ARCS 2006*, W. Grass, B. Sick, and K. Waldschmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3894, pp. 311–325.

[9] A. E. I. Brownlee, N. Burles, and J. Swan, "Search-Based Energy Optimization of Some Ubiquitous Algorithms," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 188–201, Jun. 2017.

[10] B. Fernandes, G. Pinto, and F. Castor, "Assisting Non-Specialist Developers to Build Energy-Efficient Software," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 158–160.

[11] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "On reducing the energy consumption of software: From hurdles to requirements," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3382494.3410678

[12] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps," p. 1, 2018.

[13] E. Moreira, F. F. Correia, and J. Bispo, "Overviewing the liveness of refactoring for energy efficiency," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. Porto Portugal: ACM, Mar. 2020, pp. 211–212.

[14] E. Iannone, F. Pecorelli, D. D. Nucci, F. Palomba, and A. D. Lucia, "Refactoring android-specific energy smells: A plugin for android studio," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 451–455. [Online]. Available: https://doi.org/10.1145/3387904.3389298

[15] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "Multi-objective optimization of energy consumption of guis in android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3241742

[16] E. G. Daylight, T. Fermentel, C. Ykman-Couvreur, and F. Catthoor, "Incorporating energy efficient data structures into modular software implementations for internet-based embedded systems," in *Proceedings of the 3rd International Workshop on Software and Performance*, ser. WOSP '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 134141. [Online]. Available: https://doi.org/10.1145/584369.584390

[17] N. Hunt, P. S. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pp. 63–70, 2011.

[18] I. Manotas, L. Pollock, and J. Clause, "Seeds: A software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 503514. [Online]. Available: https://doi.org/10.1145/2568225.2568297

[19] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, Oct. 2016, pp. 20–31.

[20] R. Pereira, P. Simão, J. Cunha, and J. a. Saraiva, "Jstanley: Placing a green thumb on java collections," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 856859. [Online]. Available: https://doi.org/10.1145/3238147.3240473

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

[22] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the code #1: The effective impact of code refactorings on software energy consumption," in *In Proceedings of the 16th International Conference on Software Technologies*, 2021, pp. 34–46.

[23] Jae-Jin Park, Jang-Eui Hong, and Sang-Ho Lee, "Investigation for Software Power Consumption of Code Refactoring Techniques," in *SEKE*, 2014.

[24] M. Kumar, Y. Li, and W. Shi, "Energy consumption in Java: An early experience," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. Orlando, FL: IEEE, Oct. 2017, pp. 1–8.

[25] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 1–53, Aug. 2016.

[26] W G P Silva, Lisane Brisolara, U. B. Corrêa, and L. Carro, "Evaluation of the impact of code refactoring on embedded software efficiency," *Unpublished*, 2010.

[27] R. Pereira, T. Carcao, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Helping Programmers Improve the Energy Efficiency of Source Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 238–240.

[28] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 316–331.

[29] P. Karabyn, "Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison," p. 59, 2019.

[30] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for Energy Efficiency: A Reflection on the State of the Art," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. Florence, Italy: IEEE, May 2015, pp. 29–35.

[31] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat, and L. Seinturier, "Taming energy consumption variations in systems benchmarking," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3647. [Online]. Available: https://doi.org/10.1145/3358960.3379142

[32] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, Mar. 2018.

[33] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 455470. [Online]. Available: https://doi.org/10.1145/2989081.2989088

[34] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser, "Benchmarking Crimes: An Emerging Threat in Systems Security," *CoRR*, vol. abs/1801.02381, 2018.

[35] Y. Wang, D. Nörtershäuser, S. Le Masson, and J.-M. Menaud, "Potential effects on server power metering and modeling," *Wireless Networks*, Nov. 2018.