Comprehending Energy Behaviors of Java I/O APIs

Gilson Rocha
Federal University of Pará
Belém, Pará
grsilva@ufpa.br

Fernando Castor
Federal University of Pernambuco
Recife, Pernambuco
castor@cin.ufpe.br

Gustavo Pinto
Federal University of Pará
Belém, Pará
gpinto@ufpa.br

***Abstract*—*Background*: APIs that implement I/O operations are the building blocks of many well-known, non-trivial software systems. These APIs are used for a great variety of programming tasks, from simple file management operations, to database communications and implementation of network protocols. *Aims*: Despite their ubiquity, there are few studies that focus on comprehending their energy behaviors in order to aid developers interested in building energy-conscious software systems. The goal of this work is two-fold. We first aim to characterize the landscape of the Java I/O programming APIs. After better comprehending their energy variations, our second goal is to refactor software systems that use energy inefficient I/O APIs to their efficient counterparts. *Method*: To achieve the first goal, we instrumented 22 Java micro-benchmarks that perform I/O operations. To achieve our second goal, we extensively experimented with three benchmarks already optimized for performance and five macro-benchmarks widely used in both software development practice and in software engineering optimization research. *Results*: Among the results, we found that the energy behavior of Java I/O APIs is diverse. In particular, popular I/O APIs are not always the most energy efficient ones. Moreover, we were able to create 22 refactored versions of the studied benchmarks, eight of which were more energy efficient than the original version. The (statistically significant) energy savings of the refactored versions varied from 0.87% up to 17.19%. More importantly, these energy savings stem from very simple refactorings (often touching less than five lines of code). *Conclusions*: Our work indicates that there is ample room for studies targeting energy optimization of Java I/O APIs.**

I. Introduction

According to a recent survey with smartphone users, longer battery life is the most desired smartphone feature [33]. This concern does not pertain only to battery-drive devices. Another survey evidenced that 24% of software engineers who develop applications for traditional PCs frequently have requirements about energy usage [22]. It is thus not a coincidence that, in recent years, researchers worldwide turned their attention to better understand the root causes that could lead to energy leaks, hotspots, or even bugs and the eventual solutions to mitigate these problems [31], [37], [30], [6], [29]. The software engineering research, in particular, has seen flourishing empirical findings [24], [32], [22], [35], process, techniques [4], [2], [17], and tools [20], [16], [28] that aim at better supporting software engineers to understand and overcome software energy consumption issues. Unfortunately, due to the intrinsic complexity of any high-level programming language, these studies are far from covering the whole spectrum of programming language design, pragmatics, and resource usage from an energy efficiency perspective.

More specifically, the energy consumption of APIs that perform I/O programming is far from being well studied. This is particularly unfortunate due to the widespread use of such APIs, which are not only the building blocks of several low-level communication channels such as sockets or database drivers, but also the bedrock of high-level software applications that have anything to do with data storage or transmission. Still, these classes are used intensively by infrastructure software, such as servers and communication libraries.

More interesting to this research is the fact that these I/O APIs might have a relevant impact on energy consumption. As an example, as indicated in a recent related work [21], about 10% of the energy consumption of mobile applications is spent in I/O operations. In another study, a 4.29% energy saving in a server computer was observed when refactoring code that deals with I/O operations [20]. In spite of its importance, the few studies that approaches this topic do not present a comprehensive energy characterization of I/O APIs. Therefore, many relevant questions are still open, such as: do different Java I/O API implementations have different energy behavior? Do the way that we use these APIs (e.g., using for reading or writing operations) have any influence on their energy consumption? Is there any chance to save energy consumption by alternating between the different Java I/O APIs?

This work aims to reduce this knowledge gap by drawing an extensive characterization of energy behaviors of Java I/O APIs. Through a broad experimental exploration of 22 Java I/O APIs, we elucidate many interesting energy characteristics of these programming constructs that were so far unknown. For instance, while the `Files` class presents the least energy consumption, the `FileInputStream` class performs three times worse than `Files`. Moreover, we perceived that refactoring could play an important role in improving the energy consumption of software systems that leverage Java I/O APIs. Through very minor changes, we were able to improve the energy consumption of a non-trivial software system by up to 17%. This finding indicates that there is an opportunity to reduce the energy consumption of existing software systems with reasonably little effort from the developers.

The main contributions of this study are the following:

- It describes an empirical study, the first of its kind to the best of our knowledge, that correlates energy behaviors

of Java I/O APIs.

- It conducts an extensive experimental exploration that involves a combination of factors, ranging from Java I/O APIs, data sizes, Java I/O API usage characteristics. The exploration draws a landscape that involves thousands of distinct points in the experiment space.
- It performs the experiments using three different category of benchmarks: micro-benchmarks, optimized-benchmarks, and macro-benchmarks. The employed optimized and macro benchmarks are known in the literature and far from trivial.

II. Research Method

A. Research Questions

This work is motivated by the following research questions:

- **RQ1:** What is the energy consumption behavior of the Java I/O APIs?
- **RQ2:** Can we improve the energy consumption of the optimized and the macro-benchmarks by refactoring their use of Java I/O APIs?

Our first research question is exploratory in nature. To provide answers to **RQ1**, we instrumented 22 Java I/O APIs. To perform this instrumentation process, we employed "micro-bencharks", that is, benchmarks that perform a single task (e.g., reading from a file in the disk). These benchmarks were instrumented following rigorous measurements approaches, as discussed in Section II-B.

For **RQ2** we performed refactorings in the code base of the benchmarks to understand whether the proposed changes could positively impact their energy consumption. We employ what we consider to be three types of benchmarks: (i) *micro-benchmarks* are small programs (around 200 LoC); (ii) *optimized benchmarks* are similar to the micro-benchmarks in size, but are optimized for performance; and (iii) *macro-benchmarks* are full-fledged working software systems (comprising thousands of lines of code). Some of these optimized and macro-benchmarks were also the subject of several energy-related studies (e.g., [1], [18], [24], [31]). To avoid solutions that could be error-prone or omission-prone, we focused on refactorings that do not require extensive code changes (e.g., changes between Java I/O APIs that extend the same interface). These refactorings are very simple, with little risk of modifying program behavior (a threat to any refactoring approach [9]) and can be easily automated by a general-purpose tool.

B. Micro-, Optimized, and Macro-Benchmarks

In this section we provide more details about the micro-benchmarks (Section II-B1), the optimized benchmarks (Section II-B2), the macro-benchmarks (Section II-B3) employed in this study.

1) Micro-Benchmarks: The Java programming language is particularly rich when it comes to APIs that perform I/O operations. More specifically, these APIs can be grouped into four abstract classes: `java.io.OutputStream`, `java.io.InputStream`, `java.io.Reader`, and `java.io.Writer`. In particular, the classes `InputStream` and `OutputStream` implement I/O operations over a byte array, whereas the classes `Reader` and `Writer` deal with chars. Table I summarizes the studied Java I/O APIs. The data available at column "# in OSS projects" was gathered from the BOA [10] infrastructure.

These classes have been introduced in the Java programming language in its very early versions. Classes that extend `java.io.Writer` and `java.io.Reader` were introduced in the version 1.1 of the language, whereas the other ones are available since Java 1.0. Each one of the studied Java I/O APIs implements at least one method for input operations or at least one method for output operations. Although Table I lists 19 Java I/O APIs that can be found at the `java.io` package, we are not covering all existing Java I/O APIs. Our study covers all classes that extends the four abstract classes, with the exception of:

- `DataOutputStream`: This class has its `readLines()` method deprecated;
- `LineNumberInputStream` and `StringBufferInputStream`: These two classes are deprecated;
- `ObjectOutputStream`: This class focused on Java objects, limiting its usage to rather specific scenarios;
- `PipedOutputStream` and `PipedInputStream`: These two classes should be used together, limiting their usage to rather specific scenarios. More concretely, the former creates data that the latter consumes;
- `PipedReader` and `PipedWriter`: These classes follow the same design of the `PipedOutputStream` and `PipedInputStream` classes;
- `SequenceInputStream`: This class reads two inputs sequentially, limiting its usage to rather specific scenarios. More precisely, as its constructor is defined (i.e., `SequenceInputStream(InputStream s1, InputStream s2)`), it starts with `s1` and, when it is done, it moves to `s2`.

For each one of these classes, we benchmarked only one method. The remaining classes from the `java.io` package are exceptions (e.g., `IOException`) or utility (e.g., `Console`) classes. In addition to the classes presented at Table I, we also included three classes: (1) `RandomAccessFile` (RAF), introduced in Java 1.0, (2) `java.util.Scanner` (SCN), introduced in Java 1.5, and (3) `java.nio.file.Files` (FI), introduced in Java 1.7. These three classes do not extend from the aforementioned abstract classes; therefore, they implement I/O operations slightly differently from the other studied Java I/O APIs. For instance, the `Files` class implements three methods that perform input operations (`List<String> readAllLines(Path path)`, `Stream<String> lines(Path path)`, and `BufferedReader newBufferedReader(Path path)`). We benchmarked these three methods. Still, the `Scanner` class relies on two methods (`String nextLine()` and `boolean hasNext()`) that should be used to perform input operations. These 22 classes comprise our corpus of Java I/O APIs.

TABLE I: Characteristics of the studied Java I/O APIs.

Class	Acronym	Method instrumented	Extends from	Available from	# in OSS projects
BufferedWriter	BW	`void write(String str)`	java.io.Writer	JDK 1.1	4,705
FileWriter	FW	`void write(String str)`	java.io.Writer	JDK 1.1	3,353
StringWriter	SW	`void write(String str)`	java.io.Writer	JDK 1.1	2,026
PrintWriter	PW	`void write(String str)`	java.io.Writer	JDK 1.1	10,501
CharArrayWriter	CAW	`void write(String str)`	java.io.Writer	JDK 1.1	572
BufferedReader	BR	`int read()`	java.io.Reader	JDK 1.1	12,441
LineNumberReader	LNR	`int read()`	java.io.Reader	JDK 1.1	897
CharArrayReader	CAR	`int read()`	java.io.Reader	JDK 1.1	187
PushbackReader	PBR	`int read()`	java.io.Reader	JDK 1.1	779
FileReader	FR	`int read()`	java.io.Reader	JDK 1.1	1,695
StringReader	SR	`int read()`	java.io.Reader	JDK 1.1	536
FileOutputStream	FOS	`void write(byte[] b)`	java.io.OutputStream	JDK 1.0	3,541
ByteArrayOutputStream	BAOS	`void write(byte[] b)`	java.io.OutputStream	JDK 1.0	6,946
BufferedOutputStream	BOS	`void write(byte[] b)`	java.io.OutputStream	JDK 1.0	1,753
PrintStream	PST	`void print(String str)`	java.io.OutputStream	JDK 1.0	8,424
FileInputStream	FIS	`int read()`	java.io.InputStream	JDK 1.0	2,823
BufferedInputStream	BIS	`int read()`	java.io.InputStream	JDK 1.0	1,832
PushbackInputStream	PBIS	`int read()`	java.io.InputStream	JDK 1.0	688
ByteArrayInputStream	BAIS	`int read()`	java.io.InputStream	JDK 1.0	1,532

For each one of the studied Java I/O APIs, we implemented the following task: each Java I/O API that implements input operations reads a 20Mb HTML file. Similarly, the classes that implement output operations write into disk 20Mb of data in an HTML file.

2) Optimized Benchmarks: In addition to the micro-benchmarks, we also studied three benchmarks from the Computer Language Benchmark Game[1]. Although small (they have about 200 lines of code), the optimized benchmarks are intrinsically different from the micro-benchmarks due to at least two important reasons: (1) they perform sophisticated programming tasks and (2) they are designed by experts to be optimized for performance. As a consequence, these benchmarks have been employed in several optimization studies targeting programming languages [18] and virtual machines [1]. Among the benchmarks available at the Computer Language Benchmark Game repository, only three of them perform I/O operations. They are:

FASTA:[2] This benchmark groups DNA, RNA, or proteins structures. This benchmark uses the method `write(byte[] b)` from the `OutputStream` class to write sequences of characters such as "GGGATACCGTACA" in the output stream. This benchmark performs only output operations and has 329 lines of code.

K-NUCLEOTIDE:[3] This benchmark takes a DNA sequence, and counts the occurrences and their frequencies of nucleotide patterns. It receives as input the FASTA output. It uses the method `String readLine()` from the `BufferedReader` class to read the output file generated by the FASTA benchmark. This benchmark performs only input operations and has 205 lines of code.

REVERSE-COMPLEMENT:[4] It reads the output of the FASTA benchmark and creates the reverse complement of each sequence, that is a new DNA sequence that connects with the old one. In parallel, it saves the reverse complements in text files. It reads the FASTA output using the method `int read(byte b[], int off, int len)` from the `InputStream` class. To write the output, it uses the method `void write(byte b[], int off, int len)` from the `OutputStream` class. This benchmark performs input and output operations and has 292 lines of code.

3) Macro-Benchmarks: Finally, we also explored macro-benchmarks. These benchmarks are real, working software systems. We chose five macro-benchmarks: three of them from the DaCapo benchmarks suite [3], and two of them from open-source repositories. The DaCapo macro-benchmarks are the following: XALAN, FOP, and BATIK. The open source projects are the following: COMMONS-IO and PGJDBC. The DaCapo benchmarks have multiple workload configurations available, e.g., small, medium, and large. These workloads vary in terms of the input size. For the other benchmarks, we created the workloads ourselves. More details about the macro-benchmarks is provided next.

XALAN:[5] This is an XSLT processor that translates XML documents into HTML files, or other types of documents. The benchmark processes 17 XML files, which are small and have a maximum size of 40 KB. The benchmark was built to use different workloads; The workloads used were: small (repeats the process 10 times; for a total 170 XML files processed; on average, the output file has 320 KB), default (repeats the process 100 times; for a total 1,700 XML files processed; on average, the output file has 3.1 MB), and large (repeats the process 1,000 times; 17,000 XML files processed. On average, each output file has 30 MB). This benchmark has 171,908 lines

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fasta-java-5.html

[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/knucleotide-java-1.html

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/revcomp-java-8.html

[5] https://xml.apache.org/xalan-j/

of code. According to the Maven Repository, 1,030 artifacts use this benchmark[6].

FOP:[7] This benchmark converts an XSL file, which has content and formatting information, to a PDF format. The workloads used were: small (an XSL file with 66 KB, which produces an output PDF file with 28.6 KB), default (an XSL file with 360 KB, which produces an output PDF file with 866 KB), and large (an XSL file with 480 KB, which produces an output PDF file with 1.8 MB). It has 95,799 lines of code. According to the Maven Repository, 145 artifacts use this benchmark[8].

BATIK:[9] This benchmark is a toolkit for applications that want to use images in the Scalable Vector Graphics (SVG) format. The implemented benchmark renders images in a PNG format from SVG entries. Three workloads were used: small (processes one SVG file, and outputs one PNG file), default (processes three SVG files, and outputs three PNG files), large (processes seven SVG files, and outputs seven PNG files). It has 170,121 lines of code. According to the Maven Repository, 73 artifacts use this benchmark[10].

COMMONS-IO:[11] This macro-benchmark is an Apache utility library used to provide high-level I/O abstractions to third party software applications. In particular, this macro-benchmark employs the method `List<String> readLines(File, Charset)` of the `FileUtils` class, which uses the `BufferedReader` class behind the scenes. In our workload, this macro-benchmark reads sequentially each line of a 250MB file with more than 4 million lines. This benchmark has 55,580 lines of code and is a widely used by open source projects. According to the Maven Repository, 16,261 artifacts use this benchmark[12].

PGJDBC:[13] This macro-benchmark is the official PostgreSQL driver for the Java programming language. This macro-benchmark communicates with a database using the `BufferedOutputStream` class, with a fixed buffer size (8KB). The benchmark performs a bulk operation: 10,240 lines are inserted sequentially in a single table with three columns (schema: a int4, b varchar (100), and c int4). The inserted string contains random numeric values, ranging from 0 to 10,239. Since this is a bulk operation, only one commit is made after the transaction. To avoid network overhead, both client code (the one that performs the insert operations) and server code (the one that stores the database) are in the same machine. This macro-benchmark has 58,044 lines of code. According to the Maven Repository, 22 artifacts use this benchmark[14].

[6]https://mvnrepository.com/artifact/xalan/xalan
[7]https://xmlgraphics.apache.org/fop/
[8]https://mvnrepository.com/artifact/org.apache.xmlgraphics/fop
[9]https://xmlgraphics.apache.org/batik/
[10]https://mvnrepository.com/artifact/org.apache.xmlgraphics/batik-util
[11]https://commons.apache.org/proper/commons-io/
[12]https://mvnrepository.com/artifact/org.apache.xmlgraphics/batik-util
[13]https://github.com/pgjdbc/pgjdbc
[14]https://mvnrepository.com/artifact/com.impossibl.pgjdbc-ng/pgjdbc-ng

C. Experimental environment

All experiments were conducted in an Intel Core machine (i7-2670QM), called **System#1**, with 4 processors (2.20GHz) running Ubuntu Linux (version 16.04 LTS, kernel 4.4.0-112-generic) with 16GB of DDR3 1600MHz memory, and Java(TM) SE Runtime Environment, version 1.8.0-151.

All experiments were performed with no other load on the OS (except the OS processes). We measure energy consumption through the `jRAPL` [20] library. This library works as an interface to the MSR (Machine-Specific Register) module, which is available to Intel architectures that support RAPL (Running Average Power Limit). RAPL stores data regarding power usage, which can afterwards be accessed through the MSR module. Using `jRAPL`, Java programmers can use a simple method call to gather power dissipation data (P, measured in watts) over time (t, measured in seconds). Energy consumption is then calculated as $E = P \times t$.

We also took particular care to mitigate noise that could interfere in the results [1], [11]. We performed each benchmark 10 times. Since it requires some time to the Just-In-Time (JIT) compiler to identify the hot code and perform optimizations, we discarded the first three executions of the benchmarks. The data reported throughout this work is the average of the seven remaining executions. Moreover, since the garbage collector and the heap size could influence in our experiments, we fixed them accordingly to mitigate potential variations. In particular, we used the parallel garbage collector (`-XX:+UseParallelGC`). The heap size was fixed at 261 MB, minimum (`-Xms`), and 4,183 MB, maximum (`-Xmx`). No other JVM options were employed.

Replication Package All data created in this study are available for replication and reproduction purposes at: https://doi.org/10.5281/zenodo.3253349.

III. RESULTS

RQ1: Energy consumption behavior of the Java I/O APIs

Figure 1 summarizes the results for this research question. The bars represent energy consumption data, whereas the lines represent power dissipation. The overall energy consumption is the sum of CPU, UNCORE (parts of CPU that do not include processors, such as caches and interconnectors), and DRAM individual energy consumption. Figure 1-(a) shows the Java I/O APIs that implements input operations, while Figure 1-(b) shows Java I/O APIs that implements output ones. The first remarkable observation from this figure is that energy behavior of the Java I/O APIs varies greatly. Generally speaking, input operations consume more energy than output operations (on average: 96 joules vs 0.80 joules, respectively).

Java I/O APIs that perform input operations. In terms of input operations, the micro-benchmark `PushbackInputStream` is the most energy consuming one (492 joules consumed), followed by `FileInputStream` (474 joules). Analyzing the `PushbackInputStream` implementation, we perceived that this Java I/O API adds a flag in the `InputStream` that marks bytes as “not read”. Such bytes are included back in the buffer to

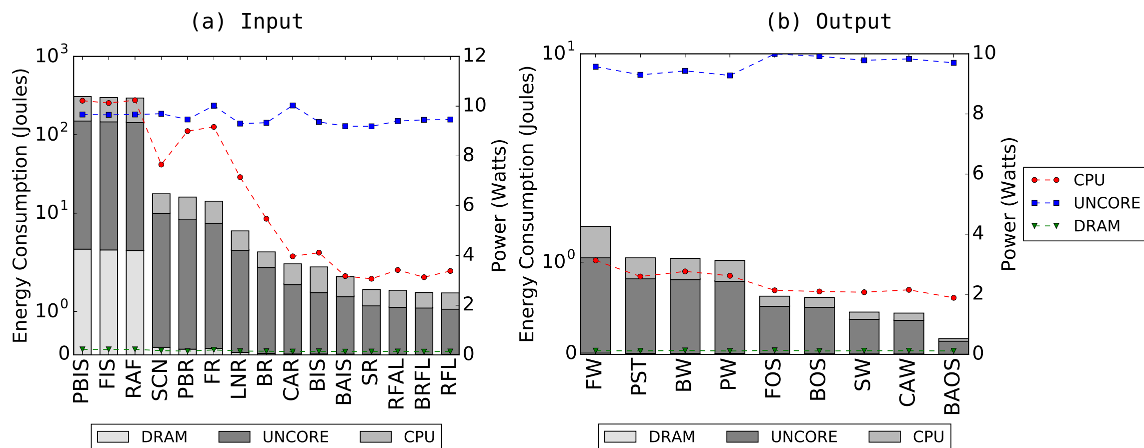


Fig. 1: Energy consumption behavior of Java I/O APIs. Energy data is presented in a logarithmic scale. For the figure on the left, PBIS stands for `PushbackInputStream`, FIS stands for `FileInputStream`, RAF stands for `RadomAccessFile`, SCN stands for `Scanner`, PBR stands for `PushbackReader`, FR stands for `FileReader`, LNR stands for `LineNumberReader`, BR stands for `BufferedReader`, CAR stands for `CharArrayReader`, BIS stands for `BufferedInputStream`, BAIS stands for `ByteArrayInputStream`, SR stands for `StringReader`, RFAL stands for `Files.readAllLines`, BRFL stands for `Files.newBufferedReader`, and RFL stands for `Files.lines`. For the figure on the right, FW stands for `FileWriter`, PST stands for `PrintStream`, BW stands for `BufferedWriter`, PW stands for `PrintWriter`, FOS stands for `FileOutputStream`, BOS stands for `BufferedOutputStream`, SW stands for `StringWriter`, CAW stands for `CharArrayWriter`, BAOS stands for `ByteArrayOutputStream`.

be read again. However, before reading the bytes, this Java I/O API also checks whether the stream is still open using the `ensureOpen()` method. This repetitive operation has the potential to be the source of this high energy consumption.

Interestingly, according to our query made at BOA [10], `FileInputStream` is heavily used in open source projects (2,823 OSS projects employ this Java I/O API). On the other hand, the `Files` micro-benchmark, which could act as a potential replacement for `FileInputStream`, is the one with the least energy consumption, when performing with its `lines` method (1,86 joules). Similar energy consumption was found when performing with other methods such as `Files.newBufferedReader` (1,90 joules) and `Files.readAllLines` (1,97 joules). In general, classes that exhibited greater energy consumption did so because their memory usage was more intensive (and therefore consumed more energy) than the other classes. In addition, methods from the `Files` class, which is part of the `java.nio` package, regularly consumed less energy than counterparts from the `java.io` package. `Files.newBufferedReader` and `Files.readAllLines` can be partially justified due to the fact that the latter uses the former in its implementation, adding to it additional behavior, such as including the read lines in a `List<String>` object. Furthermore, a careful reader would observe that the method `Files.newBufferedReader` shares a similar name to the class `BufferedReader`, although its energy behavior is slightly different. To better understand these differences, we studied the source code of these classes. We believe this happens because the `Files.newBufferedReader` method uses a different `InputStream` object provided by the `java.nio.file.FileSystemProvider` class. On the other hand, the `BufferedReader` class, specifically for this micro-benchmark, uses the `java.io.FileReader` as an instance of the `InputStream` class.

Java I/O APIs that perform output operations. Regarding the Java I/O APIs that perform output operations, the `FileWriter` class was the most inefficient one (1,55 joules consumed), followed by `PrintStream` (1,30 joules), and `PrintWriter` (1,29 joules). This behavior is particularly due to the way `FileWriter`, in particular, writes chars to file. Before writing the chars, they pass through an encoding process. The encode could change depending on the chosen charset. The code snippet presented in Figure 2 describes the use of the `encoder` object inside the `implWrite()` method (which is called by the `write()` method).

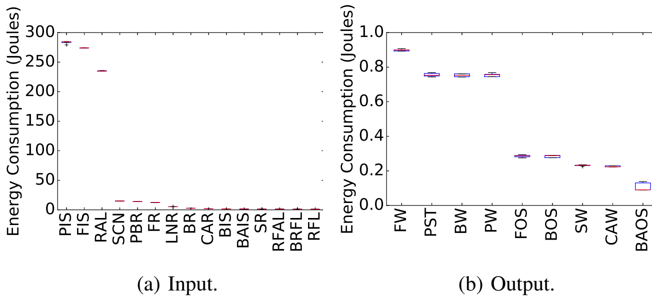
```
void implWrite(char[] v1, int v2, int v3) {
    // instance the character buffer in 'v4'
    while(v4.hasRemaining()) {
        // perform the character encode
        v5 = encoder.encode(v4, bb, false);
        // write data
} }
```

Fig. 2: A code snippet of the `StreamEncoder` class, used by the `FileWriter` class.

The `PrintStream` and the `PrintWriter`, on the other hand, work as wrappers of the `BufferedWriter` class, adding to it additional features such as efficient writing of sin-

gle characters, arrays, and strings. This behavior might explain the similar energy behavior along these Java I/O APIs (i.e., `PrintStream`, `PrintWriter`, and `BufferedWriter`). The `ByteArrayOutputStream` class, on the other hand, is the benchmark that performed the best, in terms of energy efficiency (0,18 joules). In this particular benchmark, all data is stored in the main memory, therefore, it avoids the well-known costly disk operations (e.g., seek time [38]).

To further substantiate our findings, we performed the same experiments but now varying the data size (from 1mb, 10mb, to 20mb). The energy variation of the experiments was rather small (no standard deviation was greater than 0.2 Joule). Figure 3 shows the results of the variation, using seven last samples of the 20mb configuration.



(a) Input. (b) Output.

Fig. 3: Experiments with the data size fixed at 20mb.

RQ2: Does refactoring play a role?

In this research question we aimed to understand whether refactoring could play a role at improving the energy consumption of (macro-) benchmarks that perform I/O operations. For each one of the (macro-) benchmarks, we manually refactored their code base, using the following steps:

1) We identified the instances of Java I/O APIs in variables/methods within the workload classes;
2) We refactored these instances to other Java I/O APIs that inherit from the same parent class;
3) We made sure that the refactored source code did not introduce any compiler or runtime error;
4) We benchmarked the refactored code following the same methodology depicted at Section II-C

Overall, we performed 22 manual refactorings. As we shall see in Table II, not all (macro-) benchmarks could be refactored to use other Java I/O APIs. In some cases (e.g., when experimenting with XALAN), we could only refactored to one other Java I/O API. Since there is a semantic gap between the Java I/O APIs that do not inherit from the same parent, we opted not to bridge this gap using an ad hoc solution, which could be error-prone and omission-prone. We then only refactored instances of Java I/O APIs that share the same parent class. The only exception to this rule was with the COMMONS-IO, which in the refactored code favors the "line-by-line" approach. Table II summarizes the results after applying the refactorings our benchmarks.

Refactoring the optimized benchmarks. As aforementioned at Section II-B2, not all benchmarks leverage input and output operations at the same workload. For instance, the FASTA benchmark does not employ any input operations, therefore, its input columns are all empty (represented with a ✖ symbol). Similarly, the K-NUCLEOTIDE benchmark does not perform output operations. Among the optimized benchmarks, only REVERSE-COMPLEMENT performs both input and output operations. The FASTA benchmark employed an `OutputStream`, then we refactored its code to use a `FileOutputStream`, a `ByteArrayOutputStream`, a `BufferedOutputStream`, and a `PrintStream`. In the default implementation, this benchmark used the `System.out` class to write the output in the terminal. As we can see in the Table II, the best energy consumption was achieved using the default implementation. According to the official documentation[15], this `System.out` class "*corresponds to display output or another output destination specified by the host environment or user*." Interestingly, the variable `out` is an instance of a `PrintStream`. However, differently than the `PrintStream` implementation, the output channel used by `System.out` is always open, which reduces the effort of opening and closing it.

On the other hand, the `ByteArrayOutputStream` API had the worst energy behavior, consuming 36% more energy than the default implementation. As aforementioned, the `ByteArrayOutputStream` deals with data in the main memory (avoiding delays related to disk access). For this particular benchmark, all changes made through the refactorings increased energy usage. Interestingly, even the class `BufferedOutputStream`, that was among the best ones in our micro-benchmark experiments, performed worse than the default implementation. Indeed, the `BufferedOutputStream` class introduced additional features aimed to further reduce disk access. According to the documentation, using this class, "*an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.*" However, this only happens when the buffer is full, which might not be case of the workload of this benchmark.

Moreover, when looking at the K-NUCLEOTIDE benchmark, we could observed that, again, the default implementation performed better than the alternative Java I/O APIs. In this case, the default implementation leverage the `BufferedOutputStream` class, an well-known and well-optimized version of `OutputStream`. Most interestingly, however, was the result obtained when performing with the `Scanner` class, which increased energy consumption by 60%. We observed that this particular class performs extra tasks, such as breaking its entry into tokens using a delimiter. The resulting tokens can then be converted to values of different types using utility methods (e.g., `Scanner.nextByte` or `Scanner.nextDouble`.

Finally, the REVERSE-COMPLEMENT benchmark performs both input and output operations. Its default implementation leverages the `FileInputStream` class for performing input operations, and the `FileOutputStream` class, for output ones.

[15]https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out

TABLE II: Benchmarks energy consumption data. FA stands for FASTA, KN stands for K-NUCLEOTIDE, and RC stands for REVERSE-COMPLEMENT. </> indicates the default implementation. ✖ indicates that the benchmark does not employ input/output operations. ☹ indicates that the refactoring could not be made. Green cells indicate energy improvements, whilst red cells indicate otherwise.

	optimized benchmarks			macro-benchmarks				
Classes	FA	KN	RC	XALAN	FOP	BATIK	COMMONS-IO	PGJDBC
`FileInputStream`	✖	☹	</>	</>	✖	✖	☹	✖
`BufferedInputStream`	✖	☹	+16.31%	-17.19%	✖	✖	☹	✖
`BufferedReader`	✖	</>	☹	☹	✖	✖	</>	✖
`LineNumberReader`	✖	-7.52%	☹	☹	✖	✖	-16.47%	✖
`Files.readAllLines`	✖	-0.87%	☹	☹	✖	✖	-9.5%	✖
`Scanner`	✖	+60.77%	☹	☹	✖	✖	+486.28%	✖
`PrintStream`	+19.13%	✖	+15.24%	-7.96%	+48.38%	-3.68%	✖	+811.41%
`FileOutputStream`	+29.13%	✖	</>	</>	+73.91%	</>	✖	☹
`DataOutputStream`	☹	✖	☹	☹	☹	☹	✖	+883.22%
`ByteArrayOutputStream`	+36.24%	✖	☹	☹	☹	☹	✖	☹
`BufferedOutputStream`	+19.23%	✖	+25.96%	-11.71%	</>	-3.1%	✖	</>
`System.out`	</>	✖	☹	☹	☹	☹	✖	☹

For this set of experiment, we only changed one class per execution (i.e., when experiment with different input APIs, the output one was not modified). The most surprisingly finding in this experiment was the good performance of `PrintStream`. However, in the particular case of REVERSE-COMPLEMENT and FASTA, these two benchmarks heavily employ concurrent programming techniques. According to related work, when concurrency is in the game, energy behavior is much more complex [31], [18]. Moreover, these two benchmarks employ their own buffering techniques, which are more sophisticated than the implementation available in Java I/O APIs such as `PrintStream`. In particular, all characters printed by a `PrintStream` are converted into bytes using its own default character encoding. This converting process (similar to what the `Scanner` class does) might the be root cause of the `PrintStream` energy behavior.

These initial results provide evidence that small changes in Java I/O APIs might have the potential of improving the energy consumption of benchmarks already optimized for performance. The relatively small effort placed in these tasks further substantiate the need of additional research along these lines. In the following we revisit the experiments, but now considering the macro-benchmarks.

Refactoring the macro-benchmarks. Among the macro-benchmarks FOP, BATIK, and PGJDBC do not employ input operations (similarly, COMMONS-IO does not employ output operations). For these macro-benchmarks, our refactorings are restricted to the counterpart operations. We performed 11 refactorings on the macro-benchmarks. We were able to improve the energy consumption in six instances of these refactorings: with XALAN when performing input and output operations, with BATIK when performing output operations, and with COMMONS-IO, when performing input operations. The right hand side of Table II summarizes the results. Generally speaking, the refactorings that changed the Java I/O API to one instance of the `Buffered` family of classes improved the energy consumption of the macro-benchmarks (up to 17%, when performing with XALAN).

We did not achieve any energy saving when performing with FOP and PGJDBC (both perform output operations). In these two particular macro-benchmarks, the default implementation employed the `BufferedOutputStream`, which had a good energy behavior in the micro-benchmarks. Moreover, one interesting observation is that the refactoring to the `PrintStream`, a perceived energy inefficient Java I/O API, had the potential of increasing energy usage in more than 811% on the PGJDBC macro-benchmark, and an increase of 48% on FOP.

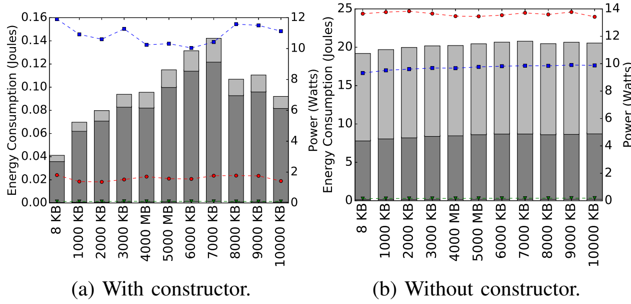
IV. FURTHER ANALYSIS

In this section we asked (and provided answers to) additional questions regarding the use of Java I/O APIs.

A. *Does the buffer size matter?*

Since the buffer size is one readily available tuning knob of the Java I/O APIs, we now conduct additional experiments varying over it. In this particular experiment, we replayed the KN experiment, using the `BufferedInputStream` Java I/O API, now varying buffer sizes from 8kb (default) to 10,000 KB. The file input was fixed on 256 MB. Figure 4 on the left shows the results. As one can see, there is a very small variation when performing the experiments with different buffer sizes (the best energy consumption was 19.67 joules, whereas the worst was 20.94 joules). Afterwards, we conducted another round of experiments, but now without the use of a constructor (the chart on the right Figure 4 used the constructor `(InputStream inputFile, int bufferSize)` to instantiate the `BufferedInputStream` object). The result without the constructor is present at the left of Figure 4. As we can see, the constructor places an energy toll in the experiment. Without the constructor, the best energy consumption was 0.04 joules (505 times lower than with the constructor). More interestingly, is the fact that we could now perceive that there is, indeed, an energy variation when using different buffer sizes. However, this variation is rather small and often shadowed when the constructor is employed. When studying the source code of the constructor, we perceived that it is

responsible for opening the input file, which consumes much more energy than the task itself.



(a) With constructor. (b) Without constructor.

Fig. 4: Additional experiments with the buffer size. Figure on the left shows the results while using a constructor, whereas figure on the right does not use a constructor.

B. Does the input size matter?

For this set of experiments, we varied the input size of the DaCapo benchmarks. As aforementioned in Section II-B3, the DaCapo benchmarks provide mechanism to vary the workload, including small, default, and large options. The results varying the workload are present at Figure 5. For each workload, we present up to four bars, which represents the results using different Java I/O APIs. These results corroborate with our initial results presented at Table II. That is, overall, the Java I/O API that performed the best in the default workload was also the one that performed the best in the other workloads.

C. Are the energy improvements statistically significant?

Here we moved one step further to understand whether the energy savings are also statistically significant. In this set of experiments, in order to gather more samples to test, we replayed the refactorings that yield energy savings (the green cells at Table II), but now iterating over them 100 times (instead of 10). We then used the non-parametric Mann-Whitney-Wilcoxon (MWW) test [39] to test whether the difference among the two transformed versions is statistically significant, and the Cliff's Delta [7], a non-parametric effect size measure for ordinal values, to measure the effect size. We interpret effect sizes e as $e < 0.1$ very small, $0.1 \leq e < 0.3$ small, $0.3 \leq e < 0.5$ medium, $0.5 \leq e < 0.7$ large, $0.7 \leq e < 0.9$ very large, and $0.9 \leq e < 1$ nearly perfect. Table III shows the results. As we can see, all the refactorings are statistically significant ($p \leq 0.05$). Still, all of them had a negative effect size, which indicates that the refactored version indeed consumed less energy.

1) Are the energy improvements platform-independent?: The results reported so far in this study are based on a single machine, **System#1**. To better exploit the external validity of our results, we replayed the DaCapo experiments in another machine (called **System#2**), with the following configuration: an Intel Xeon machine (ES-2660) with 40 processors (2.20GHz) running Ubuntu Linux (version 14.04 LTS, 3.19.0-25-generic kernel) with 251GB of main memory, and Java (TM) SE Runtime Environment, version 1.8.0-151.

TABLE III: Statistical Analysis.

Benchmark	Class	Effect Size	p-value
KNUCLEOTIDE	Files.readAllLines	-0.5496296 (large)	1.926e-10
KNUCLEOTIDE	LineNumberReader	-0.93 (large)	< 2.2e-16
XALAN	BufferedInputStream	-1 (large)	< 2.2e-16
XALAN	BufferedOutputStream	-1 (large)	< 2.2e-16
XALAN	PrintStream	-0.8282716 (large)	< 2.2e-16
BATIK	BufferedOutputStream	-0.2032099 (small)	0.01861
BATIK	PrintStream	-0.1982716 (small)	0.02168
COMMONS-IO	LineNumberReader	-0.351358 (medium)	4.707e-05
COMMONS-IO	Files.readAllLines	-0.3996296 (medium)	3.673e-06

We decided to focus only on the DaCapo macro-benchmarks because they are less prone to performance variations and are more likely to mimic a real user case experience. Table IV shows the results. As we can see, the most energy-efficient Java I/O APIs found at **System#1** were not always the ones found at **System#2**. Only in the FOP benchmark that the results coincided for the two environments. It is interesting to note in **System#1**, the classes `BufferesInputStrem` and `BufferedOutputStream` appear as the most energy efficient choices. Although the same does not hold true for **System#2**, we observed that these classes (`BufferesInputStream` and `BufferedOutputStream`) are, indeed, the second most energy-efficient in the XALAN and BATIK benchmarks, with very minor energy variations.

TABLE IV: Comparing the energy results obtained in **System#1** and **System#2**. Green cells indicate that the result matches, whilst Red indicate that does not matches.

	System#1			System#2		
	SMALL	DEFAULT	LARGE	SMALL	DEFAULT	LARGE
XALAN	BIS	BIS	BIS	FIS	FIS	FIS
FOP	FOS	BOS	BOS	BOS	BOS	BOS
BATIK	PST	PST	BOS	FOS	FOS	FOS

D. Are the most used Java I/O APIs the most energy efficient?

One interesting observation of this study is that not always the most popular Java I/O API is also the most energy efficient one. Consider the `BufferedReader` class, which is the most used Java I/O API in the OSS stored by BOA [10]: it was employed in 12,441 OSS projects. However, in terms of energy consumption, `BufferedReader` performs just ordinarialy: it is the 9th best energy behavior among the 15 Java I/O APIs that perform input operations. Similarly, the `PrintWriter` class is the second most used Java I/O API (employed in 10,501 projects), but has the fourth worst energy consumption among classes that implement writing operations. As another example, the `Scanner` class, which is fourth most used Java I/O API (7,970 OSS projects employ it), has the fourth worst energy consumption. Indeed, when we refactored the optimized and macro-benchmarks to use `Scanner`, we noted an energy drain of up to 486%.

1) How many source code lines are modified in the refactoring transformations?: Overall, our refactorings changed 3–5 lines of code. However, in same cases which the Java I/O APIs share the same parent class, the refactoring changed just

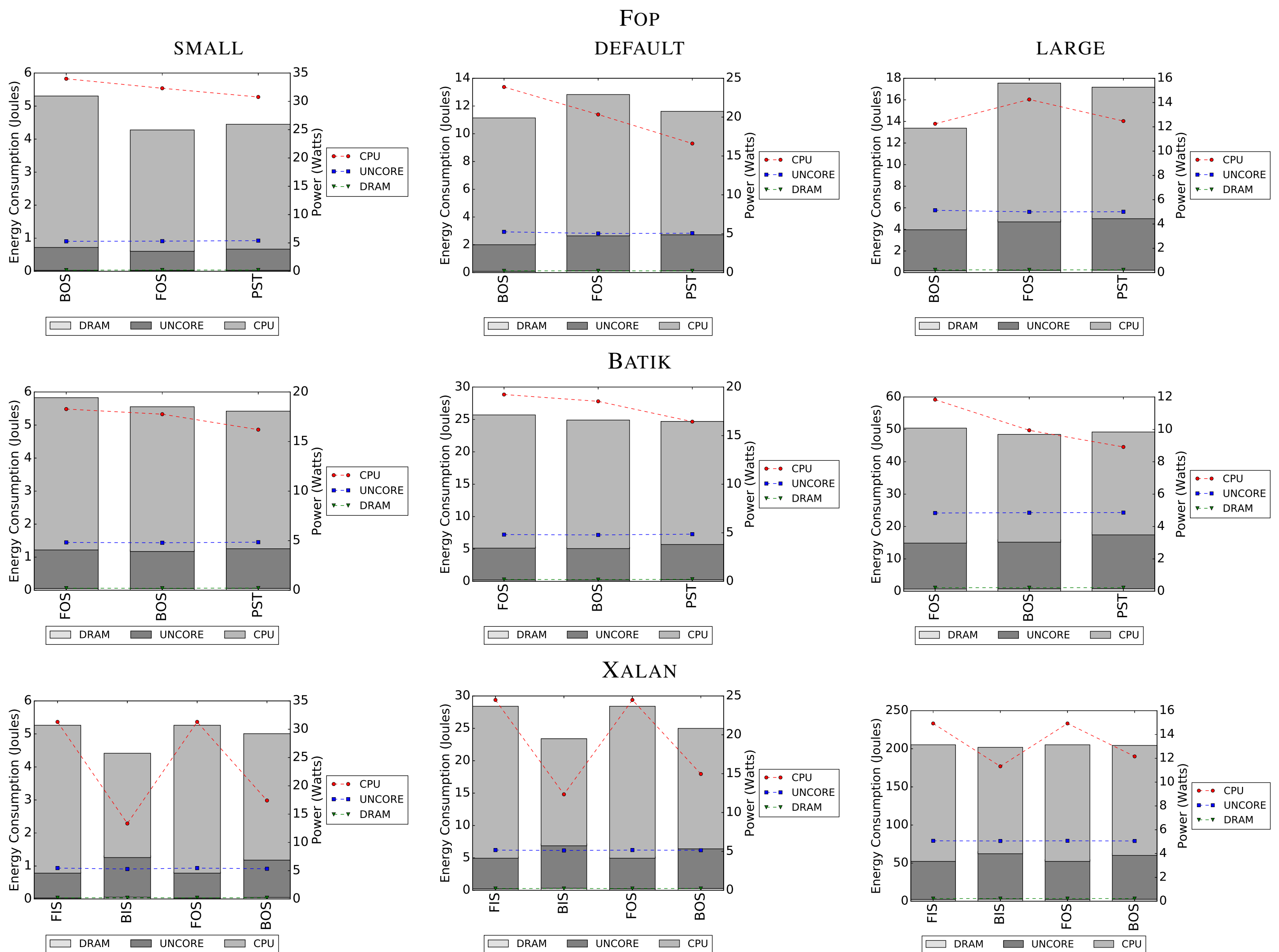


Fig. 5: Energy behavior when varying the workload of the DaCapo macro-benchmarks.

a single line of code. The Listing 1 shows a representative example.

```
  static byte[] read(InputStream is) {
-     BufferedReader in = new
-     BufferedReader(new InputStreamReader(is,
+     LineNumberReader in = new
+     LineNumberReader(new InputStreamReader(is,
      StandardCharsets.ISO_8859_1));
      while ((line = in.readLine()) != null) {
        // do work
    }
    // perform read operations
    return toCodes(bytes, position);
  }
```

Listing 1: A snippet of the `Knucleotide` before and after the refactoring to `LineNumberReader`.

V. Limitations and Threats to Validity

First, this work is limited to Java classes that reside in the `java.io` package. This package contains classes that implement I/O operation by default. However, other classes exist in other packages, as well as in third-party software libraries. Although we manually added other classes, we certainly did not explore all possible Java I/O APIs. Second, although we conducted additional experiments with some "tuning knobs", such as the buffer size and the input size, some Java I/O APIs have constructors with extensive configurations. Similarly, we did not explore concurrency characteristics of the benchmarks and Java I/O APIs. Experimenting with other configurations is a combinatorial problem, and can be the subject of follow up works (e.g., treating energy consumption as an optimization problem). Another limitation is regarding to the use of the BAOS Java I/O API. This Java I/O API reads data from the main memory, instead of the disk, which is how the other Java I/O APIs do. This behavior made BAOS very energy efficient,

when compared to the other Java I/O APIs. However, when data is in the disk, one should use BAOS with other Java I/O API, which may increase its energy consumption.

Third, we are limited by our selection of micro-, optimized, and macro-benchmarks. Nonetheless, our corpus spans a wide spectrum of benchmarks, from simple input/output operations, to benchmarks already optimized for performance, to macro-benchmarks used in daily computer programming tasks. Fourth, we performed source code modifications in the optimized and macro-benchmarks. One concern that one may raise is whether our modifications could impact on the correctness of the benchmarks. To mitigate this concern, we manually compared the outputs generated by the default implementations and the ones generated by our modifications, and they were all the same. However, a thoroughly analysis on this regard is left for future work.

Fifth, we conducted our experiments following experimental systems research guidelines (e.g., [1]). As a consequence, our environment might not represent a traditional developer workstation, which often has several running process in the CPUs. Finally, one may argue that our approach for averaging the last seven execution of the benchmarks would be sufficient to draw reliable conclusions. This threshold was chosen based on a comprehensive related work [31]. However, Figure 6 provides additional light on this matter. In this figure, we present the energy consumption of the 10 executions of the `Scanner` benchmark. As one can see, the energy consumption roughly plateau in the last executions.

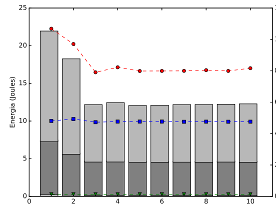


Fig. 6: Energy variation of 10 execution for the `Scanner` benchmark.

VI. Related Work

Most of the existing software empirical research has focused on analyzing the relationship between individual characteristics of an application and energy consumption. Examples of such characteristics include data structures [8], [13], [23], VM services [5], cloud offloading [14], code obfuscation [36], and design patterns [34], [15]. Such research serves as a guideline for future energy-aware application programmers.

Lima et al. [18] analyzed Haskell's thread-management constructs. The authors observed that by replacing uses of Haskell's default thread primitive, `forkIO`, by an alternative, `forkOn`, that binds the created thread to a specific processor while requiring modifications to a single line of code per use of `forkIO`, exhibited lower energy consumption in most of the experiments. Pathak *et al.* [26] categorized energy bugs through analyzing the posts from four online forums. They produced a comprehensive taxonomy ranging from battery problems, SIM card problems, OS configuration problems, to no-sleep bugs. Another study by Pathak *et al.* [27] presented an investigation aiming to understand the root causes for energy consumption problems in mobile applications. Linares-Vasquez *et al.* [19] investigated Android API usage patterns that can potentially consume high energy consumption. The authors observed that while some anomalous energy consumption is unavoidable, some can be avoided by using certain categories of Android APIs and patterns. Oliveira et al. [24] looked at the energy footprint of different programming languages used to develop Android applications. The results showed a hybrid approach (i.e., using more than one programming language to develop an app) have the potential of increasing energy efficiency. In some scenarios, small modifications (less than 10% of the lines of code of an application) can result in significant reduction in energy consumption.

There are several works that focus on the energy consumption of Java collections. Pinto et al [32] compared the energy consumption of thread safe and non-thread safe collections. Hassan et al [12] presented an empirical study over 17 collections, experimented with six applications (general purposes libraries and including mobile apps). In terms of tools, Pereira and colleagues [28] proposed a tool called jStanley that automatically finds collections that could be replaced by other in other to improve their energy usage. More recently, Oliveira [25] presented CT+, a tool that statically analyses collections usage and suggests energy efficient recommendations. This tool was experimented with 40 different collections.

These studies share a common finding: simple changes can reduce energy consumption considerably. In our work, we focus on Java I/O APIs, which is an direction that has been so far little explored. Moreover, our findings corroborate with the literature in the sense that refactorings that perform minor modifications can yield relevant energy savings.

VII. Conclusions

In this work we conducted an extensive energy categorization of 22 Java classes that implement I/O operations. After an initial understanding of the energy landscapce of Java I/O APIs, we then refactored optimized and macro-benchmarks in order to investigate whether we could achieve energy improvements by altering between Java I/O APIs. Among our findings, we observed that the energy behavior of Java I/O APIs varies greatly (and not always a very popular Java I/O API is also the most energy efficient one). In another round of experiments, we refactored 22 instances of Java I/O APIs used in optimized and macro-benchmarks. These refactorings taught us two important lessons: first, we could indeed improve energy consumption of non-trivial working systems by changing few lines of Java I/O APIs code; second, not always the Java I/O API that yield energy savings in one benchmark will also lead to energy savings in another benchmark. More study is still needed to better comprehend the specific scenarios and limitations of our proposed transformations.

For *future work* we plan to replay the experiments in a mobile platform. We also plan to extend the set of Java I/O APIs, covering eventual third-party IO APIs. Finally, based on our findings, we plan to create tools that could help developers (1) to detect which Java I/O API they should use or (2) to propose automatic refactorings to green Java I/O API.

Acknowledgments. This research was partially funded by CNPq/Brazil (304755/2014-1, 406308/2016-0, 465614/2014-0) and FACEPE/Brazil (APQ- 0839-1.03/14, 0388-1.03/14, 0592-1.03/15).

REFERENCES

[1] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, Oct. 2017.

[2] T. Bartenstein and Y. D. Liu. Green streams for data-intensive software. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 532–541, 2013.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[4] A. Canino, Y. D. Liu, and H. Masuhara. Stochastic energy optimization for mobile GPS applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 703–713, 2018.

[5] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, 2012.

[6] S. A. Chowdhury, S. D. Nardo, A. Hindle, and Z. M. J. Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, 23(3):1422–1456, 2018.

[7] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, Nov. 1993.

[8] 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*, WOSP '02, pages 134–141, 2002.

[9] D. Dig and R. Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, Mar. 2006.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431, 2013.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

[12] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *ICSE*, pages 225–236, 2016.

[13] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '11, pages 63–70, 2011.

[14] Y. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, pages 170–179, 2013.

[15] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, 2014.

[16] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 78–89, 2013.

[17] D. Li, A. H. Tran, and W. G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 527–538, 2014.

[18] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *SANER*, pages 517–528, 2016.

[19] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *MSR*, pages 2–11, 2014.

[20] K. Liu, G. Pinto, and Y. D. Liu. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 316–331, 2015.

[21] 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, Shanghai, China, September 17-22, 2017*, pages 444–455, 2017.

[22] 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 *ICSE*, pages 237–248, 2016.

[23] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE*, pages 503–514, 2014.

[24] W. Oliveira, R. Oliveira, and F. Castor. A study on the energy consumption of android app development approaches. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 42–52, 2017.

[25] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto. Recommending energy-efficient java collections. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, pages 160–170, 2019.

[26] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, 2011.

[27] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *EuroSys*, pages 29–42, 2012.

[28] R. Pereira, P. Simão, J. Cunha, and J. Saraiva. jstanley: placing a green thumb on java collections. In *International Conference on Automated Software Engineering, ASE 2018, Montpellier*, pages 856–859, 2018.

[29] G. Pinto, A. Canino, F. Castor, G. H. Xu, and Y. D. Liu. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 765–775, 2017.

[30] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 22–31, 2014.

[31] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, pages 345–360, 2014.

[32] 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 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 20–31, 2016.

[33] S. Sabin. Smartphone owners prefer simple features like battery life, durability, camera quality. https://morningconsult.com/2018/11/15/smartphone-owners-prefer-simple-features-like-battery-life-durability-camera-quality/, November 2018. Accessed: 2019-02-10.

[34] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. E. Kiamilev, L. L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *GREENS*, pages 55–61, 2012.

[35] C. Sahin, L. L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 36, 2014.

[36] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 131–140, 2014.

[37] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W. G. J. Halfond, and J. Clause. How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588, 2016.

[38] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 24–24, Berkeley, CA, USA, 2009. USENIX Association.

[39] D. Wilks. *Statistical Methods in the Atmospheric Sciences*. Academic Press. Academic Press, 2011.