Comprehending Energy Behaviors of Java I/O APIs

Gilson Rocha  Fernando Castor  Gustavo Pinto

To appear at esem 2019
University of... What?
University of... What?
University of... What?
University of... What?
University of... What?
University of... What?

Federal University of Pará (UFPA)

61 years
40K+ students
800+ professors

@gustavopinto
Uh oh
Go away bad apps!

Respect by battery!

We need energy efficient apps!
NOW!
Hello all!

Been using Waze for about two weeks now and I really like it. Haven't been stuck in any traffic jam during this time, like I normally do this time of year, but I have received a few warnings from Waze and driving alternative routes with great success. Thank You for that!

But, how come Waze uses so much battery?

I have a SonyEricssonArCs and the SonyEricsson car charger is the only one I've tried this far that can cope with the energy hungry Waze app. All other chargers, 4 of them, can't deliver enough current to keep the phones battery from draining when running Waze. The phone itself gets very hot and from what I've read on the net this isn't a problem just for me.

I also use another similar app in my Xperia and that is the Geocaching app "Neongeo". Neongeo, like Waze, uses GPS, a-GPS, maps, shows me moving around on the map, live internet update and keeps the phones screen always on. Still, it consumes what seems like much less power than Waze and the phone does not get hot at all.

Why?
I have no idea on how to make this code more energy efficient 😢
Source of Java I/O APIs

**Class Writer**
java.io
java.lang.Object
java.io.Writer

All Implemented Interfaces:
Closeable, Appendable, AutoCloseable

Direct Known Subclasses:
BufferedWriter, CharArrayWriter, FilterWriter, OutputStreamWriter, PipedWriter, PrintWriter, StringWriter

```java
public abstract class Writer
extends Object
implements Appendable, Closeable, Flushable

Abstract class for writing to character streams. The only methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Since:
JDK 1.1
```

**Class Reader**
java.io
java.lang.Object
java.io.Reader

All Implemented Interfaces:
Closeable, AutoCloseable, Readable

Direct Known Subclasses:
BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

```java
public abstract class Reader
extends Object
implements Readable, Closeable

Abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Since:
JDK 1.1
```

**Class InputStream**
java.io
java.lang.Object
java.io.InputStream

All Implemented Interfaces:
Closeable, AutoCloseable

Direct Known Subclasses:

```java
public abstract class InputStream
extends Object
implements Closeable

This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

Since:
JDK 1.0
```

**Class OutputStream**
java.io
java.lang.Object
java.io.OutputStream

All Implemented Interfaces:
Closeable, Flushable

Direct Known Subclasses:
ByteArrayOutputStream, FileInputStream, FilterOutputStream, InputStream, ObjectOutputStream, ObjectOutput, ObjectOutputStream, PipedOutputStream

```java
public abstract class OutputStream
extends Object
implements Closeable, Flushable

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

Since:
JDK 1.0
```
Source of Java I/O APIs

**Class Writer**
java.io

```java
Class Writer
java.lang.Object
java.io.Writer

All Implemented Interfaces:
  Closeable, Flushable, Appendable, AutoCloseable

Direct Known Subclasses:
  BufferedWriter, CharArrayWriter, FilterWriter, OutputStreamWriter, PipedWriter, PrintWriter, StringWriter
```

**Class Reader**
java.io

```java
public abstract class Reader
extends Object
implements Readable, Closeable

All Implemented Interfaces:
  Closeable, AutoCloseable, Readable

Direct Known Subclasses:
  BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader
```

**Class InputStream**
java.io

```java
Class InputStream
java.lang.Object
java.io.InputStream

All Implemented Interfaces:
  Closeable, AutoCloseable

Direct Known Subclasses:

public abstract class InputStream
extends Object
implements Closeable

The abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

Since:
  JDK1.0
```

**Class OutputStream**
java.io

```java
public abstract class OutputStream
extends Object
implements Closeable, Flushable

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

Since:
  JDK1.0
```
Source of Java I/O APIs

100K+ projects use Java IO APIs
(as of sept 2015)
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

try {
    StringBuilder sb = new StringBuilder();
    String line = reader.readLine();

    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = reader.readLine();
    }
    String everything = sb.toString();
} finally {
    br.close();
}
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

try {
    StringBuilder sb = new StringBuilder();
    String line = reader.readLine();
    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = reader.readLine();
    }
    String everything = sb.toString();
} finally {
    br.close();
}
try {
    StringBuilder sb = new StringBuilder();
    String line = reader.readLine();

    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = reader.readLine();
    }
    String everything = sb.toString();
}
finally {
    br.close();
}
CharArrayReader reader = new CharArrayReader(new FileReader("file.txt"));

try {
    StringBuilder sb = new StringBuilder();
    String line = reader.readLine();
    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = reader.readLine();
    }
    String everything = sb.toString();
} finally {
    br.close();
}
FilterReader reader = new FilterReader(new FileReader("file.txt"));

try {
    StringBuilder sb = new StringBuilder();
    String line = reader.readLine();

    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = reader.readLine();
    }

    String everything = sb.toString();
} finally {
    br.close();
}
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

LineNumberReader reader = new LineNumberReader(new FileReader("file.txt"));

CharArrayReader reader = new CharArrayReader(new FileReader("file.txt"));

FilterReader reader = new FilterReader(new FileReader("file.txt"));
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

LineNumberReader reader = new LineNumberReader(new FileReader("file.txt"));

CharArrayReader reader = new CharArrayReader(new FileReader("file.txt"));

FilterReader reader = new FilterReader(new FileReader("file.txt"));

---

**Similar design choices**

**Reasonable Interchangeable**

**Extremely used**
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

LineNumberReader reader = new LineNumberReader(new FileReader("file.txt"));

CharArrayReader reader = new CharArrayReader(new FileReader("file.txt"));

FilterReader reader = new FilterReader(new FileReader("file.txt"));

---

Similar design choices
Reasonable Interchangeable

Extremely used
Energy usage?

@gustavopinto
**2 environments**

**Intel CPU**: A 4-core, running Ubuntu, 2.2 GHz, 16GB of memory, JDK version 1.8.0, build 151.

**Intel CPU**: A 40-core, running Ubuntu, 2.20GHz, with 251GB of memory, JDK version 1.8.0, build 151.

Software-based energy measurement
Intel CPU: A 4-core, running Ubuntu, 2.2 GHz, 16GB of memory, JDK version 1.8.0, build 151.

**jRAPL — A framework for profiling energy consumption of Java programs**

**What is jRAPL?**

jRAPL is framework for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) support.

**But, what is RAPL?**

RAPL is a set of low-level interfaces with the ability to monitor, control, and get notifications of energy and power consumption data of different hardware levels. Originally designed by Intel for enabling chip-level power management, RAPL is widely supported in today’s Intel architectures, including Xeon server-level CPUs and the popular i5 and i7.

https://github.com/kliu20/jRAPL

# 22 Java IO APIs

## Writer
- BufferedWriter
- FileWriter
- StringWriter
- PrintWriter
- CharArrayWriter

## Reader
- BufferedReader
- LineNumberReader
- CharArrayReader
- PushbackReader
- FileReader
- StringWriter

## OutputStream
- FileOutputStream
- ByteArrayOutputStream
- BufferedOutputStream
- PrintStream

## InputStream
- FileInputStream
- ByteArrayInputStream
- BufferedInputStream
- PushbackInputStream
- ByteArrayInputStream

## Files

## RandomAccessFile

## Scanner
Micro benchmarks

Optimized benchmarks

Macro benchmarks
BufferedInputStream reader = new FileInputStream(FILE_READER);
int value = 0, fake = 0;
while ((value = reader.read()) != -1) fake = value;
reader.close();

FILE_READER = TXT
20mb
BufferedOutputStream fileWriter = new
BufferedOutputStream(new
    FileOutputStream(new File(OUT_WRITER +
    UUID.randomUUID().toString())));
fileWriter.write(data);
fileWriter.close();
Optimized benchmarks

The Computer Language Benchmarks Game

"Which programming language is fastest?"

Should we care? How could we know?

"It's important to be realistic: most people don't care about program performance most of the time."

"By instrumenting the ... runtime, we measure the JavaScript behavior of ... web applications... Our results show that real web applications behave very differently from the benchmarks..."

Source code and performance measurements available
import java.io.IOException;
import java.io.OutputStream;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class fasta {
    static final int LINE_LENGTH = 60;
    static final int LINE_COUNT = 1024;
    static final int[] WORKERS = new NucleotideSelector[] {
        new NucleotideSelector()
    };
    static final AtomicInteger IN = new AtomicInteger();
    static final AtomicInteger OUT = new AtomicInteger();
    static final int BUFFERS_IN_PLAY = Runtime.getRuntime().availableProcessors() > 1 ? Runtime.getRuntime().availableProcessors() - 1 : 1;
    static final AtomicInteger IM = new AtomicInteger();
    static final AtomicInteger IA = new AtomicInteger();
    static final AtomicInteger IC = new AtomicInteger();
    static final float ONE_OVER_IM = 1f / IM;
    static int last = 42;

    public static void main(String[] args) {
        int n = 1000;
        if (args.length > 0) {
            n = Integer.parseInt(args[0]);
        }
        for (int i = 0; i < WORKERS.length; i++) {
            WORKERS[i] = new NucleotideSelector();
            WORKERS[i].setDaemon(true);
            WORKERS[i].start();
        }
        try (OutputStream writer = System.out;) {
            final int bufferSize = LINE_COUNT * LINE_LENGTH;
            for (int i = 0; i < BUFFERS_IN_PLAY; i++) {
                lineFillALU(final byte[] sapienChars = new byte[] {'a', 'c', 'g', 't'},
                            final double[] sapienProbs = new double[]
                        {0.3029549426680, 0.1979883004921, 0.1975473066391, 0.3015094502008},
                            final float[] probs, final float[] randoms, final int charsInFullLines
                        ) {
                    super(lineLength, nChars);
                    double cp = 0;
                    final double[] iubProbs = isIUB ? iubProbs : sapienProbs;
                    char = isIUB ? iubChars : sapienChars;
                    probs = new float[iubProbs.length];
                    for (int j = 0; j < prob.length; j++) {
                        cp += iubProbs[j];
                    }
                    for (int j = 0; j < CHARS_LEFTOVER; j++) {
                        randoms = new float[nChars];
                        charsInFullLines = (nChars / lineLength) * lineLength;
                        @Override
                        public void selectNucleotides() {
                            int ox = 0;
                            float[] r;
                            int x;
                            for (x = 0; x < charsInFullLines; x++) {
                                for (k = 0; k < LINE_LENGTH; k++) {
                                    r = randoms[x++];
                                    for (m = 0; r < prob[m]; m++ | |
                                        nucleotides[x++] = char[m];
                                    }
                                    for (k = 0; k < CHARS_LEFTOVER; k++) {
                                        r = randoms[x++];
                                        for (m = 0; r < prob[m]; m++ | |
                                            nucleotides[x++] = char[m];
                                        }
                                    }
                                }
                            }
                        }
                    }
                    }
                    }
                    }
                    }
                    }
                }
            }
        }
    }
}

Fasta (325 loc)
public class Fasta {
    static final int LINE_LENGTH = 80;
    static final int LINE_COUNT = 100;
    static final int WORKERS = 100;
    static final int NUCLEOTIDESELECTOR = 100;

    public static void main(String[] args) {
        int n = 0;
        if (args.length > 0) {
            n = Integer.parseInt(args[0]);
        }
        for (int i = 0; i < WORKERS; i++) {
            WORKERS[i].setDaemon(true);
        }
        for (OutputStream writer = System.out; ) {
            try {
                for (int i = 0; i < NUCLEOTIDESELECTOR; i++) {
                    double cp = 0;
                    for (int j = 0; j < LINE_COUNT; j++) {
                        for (int k = 0; k < LINE_LENGTH; k++) {
                            for (int m = 0; m < nChars; m++) {
                                int r = randoms[i++];
                            }
                        }
                    }
                    for (int m = 0; m < nChars; m++) {
                        nucleotides[j++] = chars[m];
                    }
                }
            } finally {
                writer.close();
            }
        }
    }
};
import java.io.IOException;
import java.io.OutputStream;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class fasta {
    static final float LINE_COUNT = 80;
    static final int LINE_LENGTH = 79;
    static final float LINE_FILL_ALU = 0.3015094502008;
    static final int LINE_FILL_T_CG = 0.1979883004921;
    static final float LINE_FILL_GA = 0.3029549426680;
    static final int FILE_OVER_WD = 20;
    static final int IN = 42;
    static final float ONE_OVER_IM = 3.8775366432499;
    static final int WORKERS = 6;
    static final byte IN_INDEX = 1;
    static final byte OUT_INDEX = 2;
    static final AtomicInteger IN = new AtomicInteger(IN);
    static final AtomicInteger OUT = new AtomicInteger(OUT);
    static final AtomicInteger LINE_COUNT = new AtomicInteger(LINE_COUNT);
    static final AtomicInteger LINE_LENGTH = new AtomicInteger(LINE_LENGTH);

    public void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Usage: java Fasta <input> <output>");
            return;
        }

        try (OutputStream writer = System.out) {
            final int bufferSize = LINE_COUNT * LINE_LENGTH;
            final float sapienProbs = new float[bufferSize]();
            final float sapienProbs = new float[bufferSize]();

            int i = 0;
            for (int j = 0; j < WORKERS; j++) {
                WORKERS[j].setDaemon(true);
                WORKERS[j].start();
            }

            final float[] probs = new float[bufferSize];
            final float[] randoms = new float[bufferSize];

            for (int k = 0; k < CHARS_LEFTOVER; k++) {
                i = 0;
                for (int j = 0; j < WORKERS.length; j++) {
                    for (int m = 0; m < probs.length; m++) {
                        if (probs[m] < r; m++) {
                            i = 0;
                            for (int k = 0; k < charsInFullLines; k++) {
                                cp = isIUB ? iubChars : sapienChars;
                                final int n = Integer.parseInt(args[0]);
                                if (!Runtime.getRuntime().availableProcessors() > 1) {
                                    Runtime.getRuntime().availableProcessors() = 1;
                                }
                                final AtomicInteger IN = new AtomicInteger(IN);
                                final AtomicInteger OUT = new AtomicInteger(OUT);
                                final AtomicInteger LINE_COUNT = new AtomicInteger(LINE_COUNT);
                                final AtomicInteger LINE_LENGTH = new AtomicInteger(LINE_LENGTH);
                                final float sapienProbs = new float[bufferSize]();
                                final float sapienProbs = new float[bufferSize]();
                                System.out.println("Usage: java Fasta <input> <output>");
                                return;
                            }
                            try (OutputStream writer = System.out) {
                                final int bufferSize = LINE_COUNT * LINE_LENGTH;
                                final float sapienProbs = new float[bufferSize]();
                                final float sapienProbs = new float[bufferSize]();
                            }
                            for (int k = 0; k < WORKERS.length; k++) {
                                WORKERS[j].setDaemon(true);
                                WORKERS[j].start();
                            }
                            final float[] probs = new float[bufferSize];
                            final float[] randoms = new float[bufferSize];
                            for (int k = 0; k < CHARS_LEFTOVER; k++) {
                                i = 0;
                                for (int j = 0; j < WORKERS.length; j++) {
                                    for (int m = 0; m < probs.length; m++) {
                                        if (probs[m] < r; m++) {
                                            i = 0;
                                            for (int k = 0; k < charsInFullLines; k++) {
                                                cp = isIUB ? iubChars : sapienChars;
                                                final int n = Integer.parseInt(args[0]);
                                                if (!Runtime.getRuntime().availableProcessors() > 1) {
                                                    Runtime.getRuntime().availableProcessors() = 1;
                                                }
                                                final AtomicInteger IN = new AtomicInteger(IN);
                                                final AtomicInteger OUT = new AtomicInteger(OUT);
                                                final AtomicInteger LINE_COUNT = new AtomicInteger(LINE_COUNT);
                                                final AtomicInteger LINE_LENGTH = new AtomicInteger(LINE_LENGTH);
                                                final float sapienProbs = new float[bufferSize]();
                                                final float sapienProbs = new float[bufferSize]();
                                            }
                                            try (OutputStream writer = System.out) {
                                                final int bufferSize = LINE_COUNT * LINE_LENGTH;
                                                final float sapienProbs = new float[bufferSize]();
                                                final float sapienProbs = new float[bufferSize]();
                                            }
                                            for (int k = 0; k < WORKERS.length; k++) {
                                                WORKERS[j].setDaemon(true);
                                                WORKERS[j].start();
                                            }
                                            final float[] probs = new float[bufferSize];
                                            final float[] randoms = new float[bufferSize];
                                            for (int k = 0; k < CHARS_LEFTOVER; k++) {
                                                i = 0;
                                                for (int j = 0; j < WORKERS.length; j++) {
                                                    for (int m = 0; m < probs.length; m++) {
                                                        if (probs[m] < r; m++) {
                                                            i = 0;
                                                            for (int k = 0; k < charsInFullLines; k++) {
                                                                cp = isIUB ? iubChars : sapienChars;
                                                                final int n = Integer.parseInt(args[0]);
                                                                if (!Runtime.getRuntime().availableProcessors() > 1) {
                                                                    Runtime.getRuntime().availableProcessors() = 1;
                                                                }
                                                                final AtomicInteger IN = new AtomicInteger(IN);
                                                                final AtomicInteger OUT = new AtomicInteger(OUT);
                                                                final AtomicInteger LINE_COUNT = new AtomicInteger(LINE_COUNT);
                                                                final AtomicInteger LINE_LENGTH = new AtomicInteger(LINE_LENGTH);
                                                                final float sapienProbs = new float[bufferSize]();
                                                                final float sapienProbs = new float[bufferSize]();
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

Output

Fasta (325 loc)
import java.io.BufferedReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Container;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Scanner;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class MyProgram {
    public static void main(String[] args) throws IOException, InterruptedException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        String input = reader.readLine();

        // Process the input
        int result = processInput(input);
        System.out.println(result);
    }

    public static int processInput(String input) throws IOException, InterruptedException {
        // ... (actual implementation)
    }
}

public class ProcessInput {
    public static int processInput(String input) throws IOException, InterruptedException {
        // ... (actual implementation)
    }
}

public class MyProgram {
    public static void main(String[] args) throws IOException, InterruptedException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        String input = reader.readLine();

        // Process the input
        int result = processInput(input);
        System.out.println(result);
    }

    public static int processInput(String input) throws IOException, InterruptedException {
        // ... (actual implementation)
    }
}

public class ProcessInput {
    public static int processInput(String input) throws IOException, InterruptedException {
        // ... (actual implementation)
    }
}
public class 

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringReader;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class k-nucleotide {
    public static void main(String[] args) {
        String sequence = System.in.readLine();
        System.out.println(sequence);
    }
}
```

**Output**

```
A 30.295
T 30.151
C 19.800
G 19.754
AA 9.177
TA 9.132
AT 9.131
CT 9.091
AC 6.002
AC 6.001
AG 5.987
GA 5.984
CT 5.971
TC 5.971
GT 5.957
TG 5.956
CC 3.917
GC 3.911
GG 3.909
GG 3.902
```

**k-nucleotide (174 loc)**
```java
import java.io.Closeable;
import java.io.FileDescriptor;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class revcomp {
    public static void main(String[] args) throws Exception {
        try (Strand strand = new Strand();
            FileInputStream standIn = new FileInputStream(FileDescriptor.in);
            FileOutputStream standOut = new FileOutputStream(FileDescriptor.out);) {
            while (strand.readOneStrand(standIn) >= 0) {
                strand.reverse();
                strand.write(standOut);
                strand.reset();
            }
        }

        class Strand implements Closeable {
            private static final byte NEW_LINE = 'n';
            private static final byte ANGLE = 'A';
            private static final int LINE_LENGTH = 60;
            private static final byte[] map = new byte[128];
            static {
                for (int i = 0; i < map.length; i++) {
                    map[i] = (byte) i;
                }
                map['t'] = map['T'] = 'A';
                map['a'] = map['A'] = 'T';
                map['g'] = map['G'] = 'C';
                map['c'] = map['C'] = 'G';
                map['v'] = map['V'] = 'B';
                map['h'] = map['H'] = 'D';
                map['r'] = map['R'] = 'Y';
                map['m'] = map['M'] = 'K';
                map['y'] = map['Y'] = 'R';
                map['k'] = map['K'] = 'M';
                map['b'] = map['B'] = 'V';
                map['d'] = map['D'] = 'H';
                map['u'] = map['U'] = 'A';
            }
            private static final int NCPU = Runtime.getRuntime().availableProcessors();
            private ExecutorService executor = Executors.newFixedThreadPool(NCPU);
            private int chunkCount = 0;
            private final ArrayList<Chunk> chunks = new ArrayList<>();
            private void ensureSize() {
                if (chunkCount == chunks.size()) {
                    chunks.add(new Chunk());
                }
            }
            private boolean isLastChunk(Chunk chunk) {
                return chunk.leftIndex <= leftEndIndex;
            }
            private int ceilDiv(int a, int b) {
                return (a + b - 1) / b;
            }
            private int getSumLength() {
                int sumLength = 0;
                for (int i = 0; i < chunkCount; i++) {
                    sumLength += chunks.get(i).length;
                }
                return sumLength;
            }
        }
    }
}
```

**revcomp (296 loc)**
revcomp (296 loc)
Macro benchmarks

Xalan™
Apache Software Foundation

Apache Commons™
http://commons.apache.org/

FOP

batik
JAVA SVG TOOLKIT

PGJDBC

@gustavopinto
Macro benchmarks

- Parses XML in HTML documents
- More than 188K lines of Java code
- More than 40 FileInputStream
Macro benchmarks

> Parses XML in HTML documents
> More than 188K lines of Java code
> More than 40 FileInputStream

3 workloads

- Small: 170 files, 320kb
- Default: 1,700 files, 3 mb
- Large: 17,000 files, 30 mb
Research Questions

**RQ1:** What is the energy consumption behavior of the Java I/O APIs?

**RQ2:** Can we improve energy consumption by refactoring the use of Java I/O APIs?
Findings

(a) Input

(b) Output

@gustavopinto
RQ1: Energy behaviors

Energy (Joules)
Power (Watts)
Energy = Power * time

@gustavopinto
RQ1: Energy behaviors

Reading consume ~3x more than writing operations
RQ1: Energy behaviors

PBIS: PushBackInputStream
FIS: FileInputStream
RAF: RandomAccessFile
RQ1: Energy behaviors

**PBIS:** PushBackInputStream

**FIS:** FileInputStream

**RAF:** RandomAccessFile

**RFAL:** Files.readAllLines()

**BRFL:** Files.newBufferedReader()

**RFAL:** Files.lines()
RQ1: Energy behaviors

SCN: Scanner

The most used Java I/O API
RQ1: Energy behaviors

(a) Input

(b) Output

FW: FileWriter   BAOS: ByteArrayInputStream
RQ2: Does refactoring play a role?

1. We identified all instances of Java I/O APIs

2. We refactored these instances to other Java I/O APIs that inherit from the same parent class

3. We made sure it compile and does not raise runtime errors

4. We benchmarked again
RQ2: Does refactoring play a role?

1. We identified all instances of Java I/O APIs
2. We refactored these instances to other Java I/O APIs that inherit from the same parent class
3. We made sure it compile and does not raise runtime errors
4. We benchmarked again

22 manual refactorings performed
RQ2: Does refactoring play a role?

<table>
<thead>
<tr>
<th>Classes</th>
<th>Optimized benchmarks</th>
<th>MACRO-BENCHMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FA</td>
<td>KN</td>
</tr>
<tr>
<td>FileInputStream</td>
<td>✔️</td>
<td>☹️</td>
</tr>
<tr>
<td>BufferedInputStream</td>
<td>✔️</td>
<td>☹️</td>
</tr>
<tr>
<td>BufferedReader</td>
<td>✔️</td>
<td>&lt;️⃣</td>
</tr>
<tr>
<td>LineNumberReader</td>
<td>✔️</td>
<td>-7.52%</td>
</tr>
<tr>
<td>Files.readAllLines</td>
<td>✔️</td>
<td>-0.87%</td>
</tr>
<tr>
<td>Scanner</td>
<td>✔️</td>
<td>+60.77%</td>
</tr>
<tr>
<td>PrintStream</td>
<td>+19.13%</td>
<td>x</td>
</tr>
<tr>
<td>FileOutputStream</td>
<td>+29.13%</td>
<td>x</td>
</tr>
<tr>
<td>DataOutputStream</td>
<td>☹️</td>
<td>x</td>
</tr>
<tr>
<td>ByteArrayInputStream</td>
<td>+36.24%</td>
<td>x</td>
</tr>
<tr>
<td>BufferedOutputStream</td>
<td>+19.23%</td>
<td>x</td>
</tr>
<tr>
<td>System.out</td>
<td>&lt;️⃣</td>
<td>x</td>
</tr>
</tbody>
</table>
RQ2: Does refactoring play a role?

We improved the energy consumption in 36% of the cases (from 0.8% to 17%)
Does the buffer size matter?
Does the buffer size matter?

Stick to the default!
Does the input size matter?
Can we trust on this?

(a) Input.

(b) Output.
Can we trust on this?

(a) Input

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Class</th>
<th>Effect Size</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNUCLEOTIDE</td>
<td>Files.readLineLines</td>
<td>-0.5496296 (large)</td>
<td>1.926e-10</td>
</tr>
<tr>
<td>KNUCLEOTIDE</td>
<td>LineNumberReader</td>
<td>-0.93 (large)</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>XALAN</td>
<td>BufferedInputStream</td>
<td>-1 (large)</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>XALAN</td>
<td>BufferedOutputStream</td>
<td>-1 (large)</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>XALAN</td>
<td>PrintStream</td>
<td>-0.828716 (large)</td>
<td>&lt; 2.2e-16</td>
</tr>
<tr>
<td>BATIK</td>
<td>BufferedOutputStream</td>
<td>-0.2032099 (small)</td>
<td>0.01861</td>
</tr>
<tr>
<td>BATIK</td>
<td>PrintStream</td>
<td>-0.1982716 (small)</td>
<td>0.02168</td>
</tr>
<tr>
<td>COMMONS-IO</td>
<td>LineNumberReader</td>
<td>-0.351358 (medium)</td>
<td>4.707e-05</td>
</tr>
<tr>
<td>COMMONS-IO</td>
<td>Files.readLineLines</td>
<td>-0.3996296 (medium)</td>
<td>3.673e-06</td>
</tr>
</tbody>
</table>
Source of Java I/O APIs

22 Java I/O APIs

Writer
- BufferedWriter
- FileWriter
- StringWriter
- PrintWriter
- CharArrayWriter

Reader
- BufferedReader
- LineNumberReader
- CharArrayReader
- PushbackReader
- FileReader
- InputStreamReader
- StringReader

OutputStream
- FileOutputStream
- ByteArrayInputStream
- BufferedOutputStream
- PrintStream

InputStream
- FileInputStream
- BufferedInputStream
- PushbackInputStream
- ByteArrayInputStream

Files
- RandomAccessFile

Scanner

@gustavopinto
Source of Java I/O APIs

RQ1: Energy behaviors

PBIS: PushBackInputStream
FIS: FileInputStream
RAF: RandomAccessFile

RFAL: Files.readAllLines()
BRFL: Files.newBufferedReader()
RFAL: Files.lines()
Source of Java I/O APIs

RQ1: Energy behaviors

PBIS: PushBackInputStream
FIS: FileInputStream
RAF: RandomAccessFile

RFAL: Files.readAllLines()
BRFL: Files.newBufferedReader()
RFAL: Files.lines()

22 Java I/O APIs

RQ2: Does refactoring play a role?

We improved the energy consumption in 36% of the cases (from 0.8% to 17%)
Source of Java I/O APIs

22 Java I/O APIs

RQ1: Energy behaviors

RQ2: Does refactoring play a role?

We improved the energy consumption in 36% of the cases
(from 0.8% to 17%)

Email
Twitter
Web site

mail@gustavopinto.org