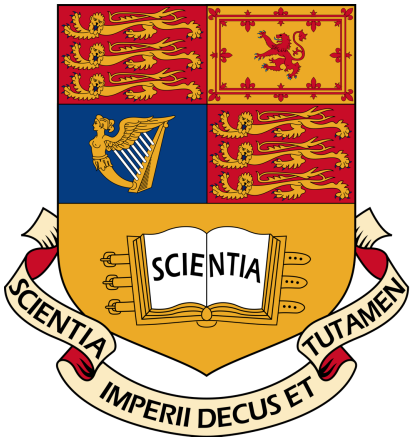
Imperial College London

Department of Computing

Correlating Bytecodes of Java Applications with Power using PowerTOP

Author:
Hong Da Koh

Supervisor:
Dr. Anandha Gopalan



Abstract

Focus of energy efficiency in software applications have been highly overlooked by developers as compared to that of hardware. The search for greater performance on algorithms might lead to a trivial increase in performance at an exponential cost of energy consumption. This report seeks to document the energy complexity behind various Java algorithms and I/O communication.

We document various investigations found from testing out the Linux power optimization tool, PowerTOP. We also document the design and implementation of a program that would allow developers to further correlate the energy complexity behind their algorithms and if the cost is worth the performance increase with PowerTOP being used as the main power estimation tool.

Our results suggest that using different search and sorting algorithms implemented manually or using Java APIs do not result in any noticeable difference in power consumption. As for I/O calls, we note that java.io.BufferedWriter is more power-efficient as compared to other forms of I/O writes such as java.io.FileWriter or the buffer-oriented java.nio package. We have also listed down several noteworthy elements of the PowerTOP power optimization tool.

Acknowledgment

I would like to thank:

- Dr. Anandha Gopalan, for his guidance and support throughout the entire project. His suggestions and feedbacks have been extremely helpful to me.
- Prof. Julie McCann, my second marker for her invaluable advice towards the direction of development for my project.
- my friends and family for their unending support throughout the project.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, Green Computing has gained importance across the world. There is a great increase in the percentage of population using computers, causing the amount of global energy consumption to increase drastically [8, 9]. Situations of energy demand exceeding supply are happening in countries like the United Kingdom [10], leading to a plethora of issues that have to be solved. Furthermore, there might even be cases of climate change as a result of this [11].

Using renewable energy could be a good alternative we can depend on for the unforeseeable future, yet this is not something we should solely fixate ourselves to. Instead, it will be more sensible to fork out different methods of saving energy. Thus, it will be sensible for actions to be taken in order to increase energy efficiency in terms of computation usage.

Many conferences such as the International Green Conference (IGCG)[1], International Conference on Green Computing and Internet of Things (ICGCIoT)[2] and International Conference on Green Computing and Engineering Technologies (ICGCET)[3] are being held either as a result of the global trends, or the advancement of technology which allows for further development in terms of energy efficiency. Regardless of it, there are indeed more conferences held as global institutes seek for more solutions to handle the issue of energy efficiency.

In fact, the current technology has seen huge improvements in this. Many household items like light bulbs, air-conditioner or fridge come with an energy label[4] which serves as a system in uniting and supporting the population towards this trend of saving energy.

That said, advancements have visibly occurred in terms of hardware energy efficiency as seen in Figure 1 [1]. As microchips get smaller and processing power gets faster, there comes a limit as to the maximum bound for how fast the processors of a computer can get while remaining within the laws of physics, marking a ceiling to Moore's Law. Nanoscience has proven to cut down the space required for fitting a number of transistors per unit area, allowing for space of manufacturers to increase the quantity of transistors to

[1]More information can be found here: `http://igsc.eecs.wsu.edu`.

[2]More information can be found here: `http://gciot-conference.org/2017/`.

[3]More information can be found here: `http://www.icgcet.org`.

[4]An example of the European Union energy label and standards: `https://ec.europa.eu/energy/en/topics/energy-efficiency/energy-efficient-products`.

increase speed. But what is more relevant to us is that as technology advances, innovations are produced in making hardware energy-efficient. Even though this is a positive phenomenon, the software aspect on energy efficiency for computers are still ignored by many as developers themselves solely seek efficiency in speed and not of energy efficiency.

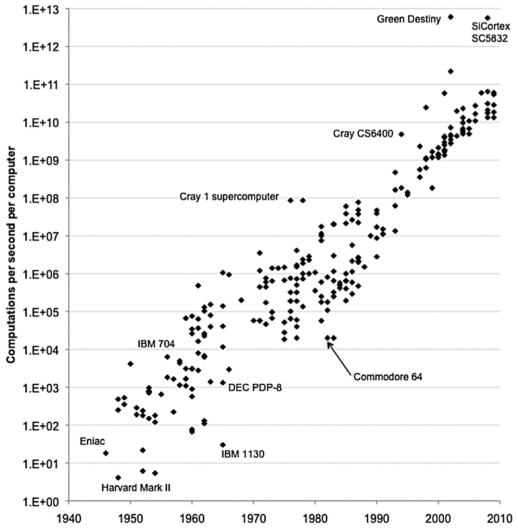


Figure 1: 'Computations per kilowatt-hour over time. These data include a range of computers, from PCs to mainframe computers and measure computing efficiency at peak performance. Efficiency doubled every 1.57 years from 1946 to 2009.' Adapted from [1].

1.1 Motivation

According to a survey done [12], most developers did not really take into account energy efficiency as they developed software applications. Whenever they discussed about energy efficiency, most of them would think of only improvement done on the hardware and not the software of computers. One reason for this might be due to the lack of knowledge on how energy efficiency could be applied on softwares.

This is further shown in another research paper [13] where the lack of knowledge in this aspect led to the requirement of using intuition to gather more information on the trade-offs between different types of coding. Thus,

energy profilers such as *eprof* [14], *eCalc* [15] and more recently, *PETrA* [16] were developed by researchers to solve this issue of improving energy efficiency on the software layer of Green Computing.

The current field of Green Computing in software remains quite primitive with no readily available solutions out in the market. Still, there have been many papers and applications available suggesting new ideas of energy measurement, especially for mobile devices. This is common within the Apple App Store[5] and Google Play[6] where applications are being developed for optimization and measurement of energy consumption on mobile phones. However, there is an apparent lack of such tools for personal computers[7] (PCs).

One reason might be because mobile phones are portable and kept small. There is a need and desire for the battery lifespan of phones to be kept as long as possible without having to charge it too frequently. Furthermore, the size of mobile phones (unlike those of laptops) limits the total amount ofpower that can be stored in the battery that is built within. Energy usage would thus be required to be more efficient, bringing a greater focus on mobile devices in the field of Green Computing. Yet this does not represent the lack of need for Green Computing to be applied on the sector of PCs.

1.2 Objectives

The main objective of this project aims to discover a relationship between energy complexity and the underlying bytecodes decompiled from a Java application. Java is currently the most popular programming language [17] and that is also the reason why it is chosen as the main working language for this project. By targeting the Java programming language, we will be able to ensure that an optimal reach of Green Computing is met.

To achieve further understanding of the relationship, a program is developed for modeling purposes via usage of a program that estimates the power consumption of the Java application under investigation. Implementation of the algorithms will be varied for further experimentation. For this project, a fully software-based solution will be used since there are hefty requirements for hardware-based solutions in terms of calibration and purchasing of

[5]An online version of App Store: `https://itunes.apple.com/us/genre/ios/id36?mt=8`.

[6]The online version of play: `https://play.google.com/store?hl=en_GB`.

[7]Referring to computers like laptops and desktops.

additional tools.

This program developed will be in the form of a plugin for the IntelliJ IDEA Integrated Development Environment (IDE)[8]. Primarily, this plugin will enhance the energy efficiency of applications written by developers through provision of a graphical representation of disassembled Bytecodes. Power consumption of each step of the disassembled bytecode instructions will be shown alongside.

This power consumption will be profiled via a software utility known as PowerTOP[9]. Due to the many variables that exist especially when power is measured on a computer, assumptions will be made and explained in the sections that follow.

As a further extension, the plugin will allow users to optimize their codes in terms of energy efficiency. This optimized code will be profiled via two different methods: either crowd-sourcing for the optimal code, or the creation of a wrapper around the developer's code for enhancement of energy efficiency.

1.3 Contributuons

Although there are many parts that made up this project, what we have done so far are as follows:

- developed a JavaFX GUI bundle that shows the power consumption and decompiled bytecodes of the developer's java package.
- Linked the JavaFX GUI to the backend system that constitutes of MongoDB[10], Jenkins[11], Gitlab[12] and a second machine with PowerTOP running on it.
- Measured the power usage for a few algorithms and different I/O calls from the *java.io* and *java.nio* packages.

[8]More relevant information can be found here: `https://www.jetbrains.com/idea/`.
[9]Relevant information can be found here: `https://01.org/powertop`.
[10]More information can be found here: `https://www.mongodb.com`.
[11]More information can be found here: `https://jenkins.io`.
[12]More information can be found here: `https://gitlab.com`.

2 Background

2.1 Power measurement

Measuring energy utilization of a computer can mean different things on different levels. The most commonly thought idea of measuring energy usually refers to the idea of measuring the electrical power consumption via an external device such as the Multimeter. Table 1 [6] as discussed by Damaševičius, et al. shows some of the commonly used methods for measuring power consumption of mobile devices.

This project will be focusing on the software profiling method. One of the main reasons why a software method was chosen is due to the cumbersome nature of getting a hardware for testing purposes. There will be many steps required just to isolate and test the power consumption for a Java application, and a software profiling method that is available on open source will be much preferred as it has been tested and calibrated by others.

Model	**Advantage**	**Disadvantage**
Hardware	Hardware-based measurements are considered to be most reliable	Noise, power loss during measurement; restriction to certain types of battery; additional hardware is required
Internal profiling software	No additional hardware device or a PC is required	Additional power consumption caused by energy logging application; restricted by functions provided by OS API and/or proprietary applications; installation problems
External software	Direct interaction with phone and additional applications are not required	Restricted by functions provided by OS API and/or proprietary applications; data communication problems

Table 1: 'Evaluation of advantages and disadvantages of energy measurement models.' Adapted from [6].

In another paper, 'ANEPROF: Energy Profiling for Android Java Virtual Machine and Applications' [18], they discussed that there are generally two

forms of profiling categories – the model-based power profiling tool and the real-measurement-based power profiling tool.

The real-measurement-based power profiling tool uses a hardware soliution by measuring the exact current and voltage consumed by the entire machine while the model-based power profiling tool is a calibration of how much power is consumed and is built on top of the real-measurement-based power profiling tool. What we are interested in then, is the model-based power profiling tool, on top of it being an external profiling software according to Table 1. Currently, there are a few papers that suggest ideas on modeling the power consumption of a machine [15, 5, 19, 16], and papers that have done experiments to classify the energy complexity of certain method calls [20, 21, 14, 22]. Even though most of these papers are geared towards mobile devices, we can still study from them.

2.1.1 *eCalc*

As presented in the paper, 'Estimating Android ApplicationsĆPU Energy Usage via Bytecode Profiling' [15], *eCalc* is designed by the researchers in the University of South California. This model designed uses an entirely software-based estimation on the execution trace of the application under review.

It requires a CPU profile to be provided by the manufacturer, such that the amount of energy consumption of every instructions could be determined. From there, the total amount of energy consumed by any software executing within the Android Operating System will be extrapolated from the data.

This model of energy profile might seem logical at first glance, however, the fact remains that there are still many other processes and interference that have to be taken into account of in order to ensure the energy profile measured suits the instructions as closely as possible, especially when we are generating a model on laptop machines instead of mobile phones. The accuracy to the ground truth values provided by *eCalc* is reasonable such that we could implement part of its method without the need to use a hardware-based solution to measure energy. Furthermore, this model serves as a check in our case to compare our measured values with.

2.1.2 *KVM enprofiler*

The paper 'An Energy Consumption Model for Java Virtual Machine' discussed how they conducted an experiment based on the Suns Microsystems K Virtual Machine (KVM) using a modified version of the energy profile *enprofiler* developed by the Embedded Systems Groups at Dortmund University as cited by Lafond & Lilius's paper [5]. The experiment they conducted focused mainly on the distribution of energy within hand-held devices, of where the majority of energy is being consumed.

They concluded that the Java virtual machine's interpreter is far ahead in energy consumption as compared to that used by the memory. This is especially crucial for us to understand. By fitting the energy consumption model in Java application into another model of energy consumption within the laptop machine, we will be able to extrapolate the results gleaned from that paper and take into account the corresponding processes within the relevant life-cycle of a Java application.

2.1.3 *eprof*

Pathak et al. had developed an energy profiler called *eprof* [14] which profiles energy consumption of a mobile phone based on 'system call tracing'. They argued that the usual utilization-based power modeling is inaccurate due to the presence of a 'tail' in the power states of those running applications [22], and that 'system calls are the only means which applications can gain access to hardware (I/O) components', using a system call tracing to estimate the energy consumption of applications will be more accurate.

Essentially, the papers by Pathak et al. are relevant to our project especially when power tracing with system calls. Understanding where the system calls are being made will be crucial for tracking down how the processes proceed and in turn, allow for the fundamental energy consumed by the application to be estimated. Although *eprof* was developed for tracing power on a hand-held device, we can fundamentally glean ideas on using system calls and finite state transitions as discussed in the paper [14, 22] on finding a suitable laptop-based power estimation tool.

2.1.4 Energy calculation in API calls

Energy can also be profiled based on API calls as seen from the papers [20, 21] where experiments are conducted on an Android machine. Similar

to the previous Section 2.1.3, through finding out the calls and sequences which consume the most energy, they are able to improve efficiency either by changing the calls or the sequences to something different altogether. [20] has done a comprehensive study on energy intensive method sequences and calls on the Android machine, allowing for other researchers to extrapolate their profilers based on that.

Primarily of note is [21] which builds on top of the work by Linares-Vásquez et al. by adding a feedback on energy used by hardware to make the estimation of energy on API calls more accurate.

Having a feedback mechanism and an energy profiler to connect both software and hardware is certainly a great idea. However, most of the profilers nowadays estimate energy consumption via energy usage on a machine not powered by the main power supply. For such a feedback mechanism to be built onto it, an entirely new energy measurement tool will have to be created for that. Still, the energy consumption of API routines in the Java language is still relevant to us.

2.1.5 Component-based Power Measurement

Chiyong et al. presented a case on estimating the energy consumption for Java systems by observing the interactions between components [19]. The paper has the perspective of estimating the total energy usage via summation of all individual components. As quoted, 'Components may comprise of a single class or a cluster of related classes'. They further decomposes the computational cost of the Java system of the classes into execution of bytecodes, native methods and the other operations.

This ideology supports what has been put forth as the main objective for this project – the decomposition of energy consumption into bytecode-sized level. Further study must be done, however, as to how the combinations of certain patterns of bytecodes will lead to the amount of energy being consumed per unit time.

2.1.6 *PETrA*

More recently, Di Nucci et al. came up with a software-based tool for estimating the energy profile on android applications [16]. Their tool, PETrA (Power Estimation Tool for Android) is built upon tools and API from the publicly

available Project Volta[13]. Project Volta is started by Google as an analysis into improving energy efficiency of Android devices. Similar to PETrA, our project will be developing a product that is based upon a third-party power estimation tool, PowerTOP.

2.1.7 Building a laptop-based model

Having reviewed several models built on hand-held devices, we will be able to use them as reference for our model, extrapolating the data to fit a model-based power profiler for laptops. The reason why there is no loss of generality in this case is because our project is targeting the Java programming language that runs using a Java Virtual Machine. This abstraction layer allows for opcodes and machine instructions to be validated from the layer of virtual machines (VMs) before translating into machine code s on the machine itself. Although there might be circumstances of differences in energy consumption between different systems (such as between Windows and Linux), several assumptions and actions will be taken to circumvent these issues and will be elaborated later.

2.2 Software utility tool for power measurement

There is a need for more detailed investigation into the amount of energy consumed on another system, and comparing it with the present methods done by other papers. Even though the method of measurement might be different (others might be using a real-measurement-based power profiling tool), there should be similarities within the amount of energy consumption measured.

2.2.1 PowerTOP

PowerTOP is a software utility tool designed by Intel[14]. It is an open-sourced Linux tool used for measurement of power consumption of the various applications and processes. This tool uses the Advanced Configuration and Power Interface (ACPI) in order to manage processes in its idle (C-State) or awake

[13]More information can be found here: `https://developer.android.com/about/versions/android-5.0.html`.

[14]Further detailed information as to what PowerTOP is can be found here: `https://software.intel.com/en-us/articles/powertop-primer-1`.

state (P-state). There are a few C-States, and they are given a number n respectively: Cn-State. The higher the numerical, the less power is being used by the CPU, but the longer the time is taken for the process to change from that state back to a working state. C0-State is typically synonymous to P-State as it represents the state where processor is working and not idling.

Through this and a few calibration done beforehand, we will be able to have a profile of energy being consumed by processes, which we can use as our benchmark in testing done.

In order for an estimation of how much energy is consumed per process, a calibration will have to be done. The percentage of energy consumed every second is obtained by sensing how much energy is being used on the battery by the system[15]. This percentage of energy consumed is further correlated by the underlying mechanism of PowerTOP, matching the transitions of machine states[16] with the total power loss measured by the system.

2.2.2 Requirement for using PowerTOP

One requirement for using this software tool is to use a laptop instead of a desktop as the power estimation is done via calculation of energy loss from the battery. After calibration, if the laptop itself is plugged into the main power supply, the power estimation shown for each processes will be a result from the previous calibration and might incur discrepancies in the actual power loss estimated since it is a result from the correlation between processor usage with the calibrated estimated power consumption.

As a result, the power consumption for each process is an estimation in itself and these values could not be taken at face value. Hence, what we can do will be to model the energy consumption into a complexity-based model, leaving aside any absolute values. Without using a hardware-based measuring tool, the absolute value of actual energy consumed will not be obtained. However, using a hardware-based solution is out of the scope of this project.

[15]To understand more about the battery and energy consumption by the machine on a Linux OS, we can use the command *upower -e*. More information can be found here: `https://askubuntu.com/questions/69556/how-to-check-battery-status-using-terminal`.

[16]This is done by ACPI of the CPU as mentioned earlier.

2.3 Clone detection tool

Part of the objective of this project requires the use of a good clone detection tool on the level of bytecodes to compare the similarities of algorithms used by developers. There are many clone detection tools available out in the market either as open source or commercial products such as Clone Digger[17] [23], and most of them serve the same purpose – finding how similar two pieces of codes are.

The paper 'Comparison and Evaluation of Clone Detection Tools' by Bellon et al. presents a comparison between clone detection tools [24]. Various methods of clone detection tools could be found from a secondary reference in that paper by Bellon et al. such as the comparison of the code's abstract syntax trees, token comparison and program dependency graph comparisons. The discussion and evaluation of these methods for clone detection is out of scope for this project. Table 2 show a basic summary and description of the different methods of clone detection.

Methods of Detection	**Description**
Textual comparison	Comparing whole lines to each other textually
Abstract Syntax Tree	Comparing the subtrees of same partition
Metric comparison	Copmaring code fragments in terms of their metric vectors
Token comparison	Comparing tokens that are formed using identifiers of source code
Program Dependency graph comparisons	Comparing graphs of dependnecies for control and data flow of a function
Bytecodes detection	Comparing the underlying bytecodes instead of higher level syntatical language

Table 2: Summary of different clone detection methods.

Keivanloo et al. [25, 2] presented a case of clone detection particularly relevant to this project on the bytecodes level using their approach known as SeByte. Figure 2 [2] shows the schematic behind how SeByte works.

[17]More information can be found here: `http://clonedigger.sourceforge.net/index.html`.

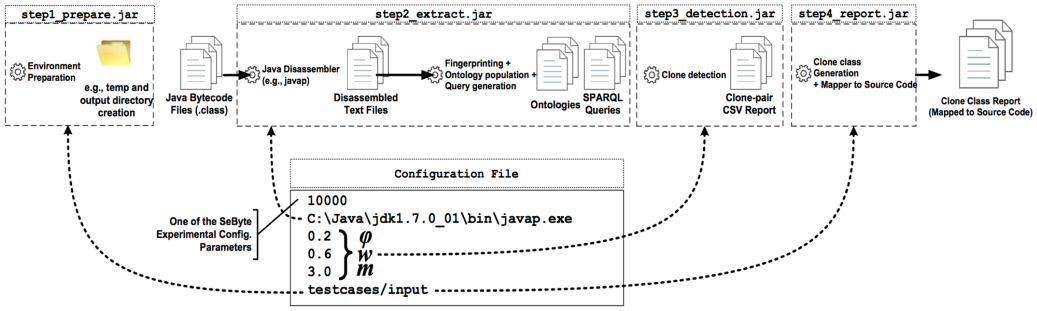


Figure 2: 'SeByte processes and the configuration file description - The execution order (left to right) and their dependency on the configuration parameters'. Adapted from [2].

2.4 System Architecture

The measurement of energy consumption for a very specific task is quite complex, especially in view of the fact that there are many other running processes and electrical components that might be using a differing percentage of energy in the system. We will try to take into account as many factors as possible in order to make the measurement accurate. To do so, we will have to understand the architecture of the system under investigation, before fitting our reasoning and modeling around it.

A Unix OS will be used, and to be exact, Lubuntu 32-bit 16.04 LTS[18]. The reason behind is because PowerTOP, the application used in measuring energy consumption and processing power usage is built for the Linux system and currently, it is not available for other OSes. As a result, Lubuntu will have to be installed as a native OS on a laptop so that the power consumed for each process could be estimated.

2.4.1 Linux Kernel

Figure 3 [3] shows the fundamental architecture behind a Linux Operating System and how it works. The user space is fundamentally the area which we interact with while most system calls are being handled within the underlying space.

Understanding the Linux architecture and how processes interact with the kernel is vital to understanding how estimation of power for Java applications

[18]The relevant information to the Lubuntu OS is located here: `http://lubuntu.net`.

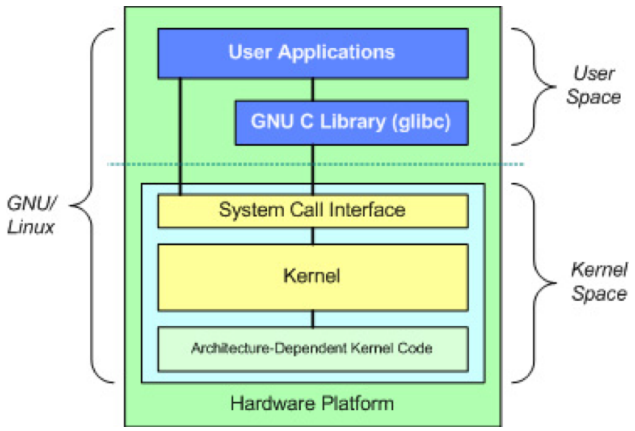


Figure 3: 'The fundamental architecture of the GNU/Linux operating system.' Adapted from [3].

are done with PowerTOP.

Typically, when we run a Java application, it will be started from the user space. Threads will be created in the processor to run the application. Anytime these threads are required to interact with the processor, a system call will have to be made. It is via this system call interface that data is passed through from the user space into the kernel space, allowing calculations to be done.

2.4.2 Power profile of processes on a Laptop

The paper 'Power Consumption Breakdown on a Modern Laptop' by Mahesri, A. and Vardhan, V. [26] shows us how much energy is being consumed during typical activities on a laptop. Usually, the power consumption of the CPU dominates for many applications when the machines are not idling. There are also other sources of power consumption on a modern laptop, such as power consumed on the Graphical User Interface (GUI).

For our case, PowerTOP primarily takes into account processes that are running in the various Cn-states. We can infer from the amount of energy used based on this configuration calibrated for the machine of all processes without requiring to use any additional hardware equipment or software, though these additional measurement might serve as a check for the estimated values obtained via PowerTOP.

2.5 Java Virtual Machine (JVM)

A complete Java project will have to be compiled before they could be run. According to Oracle, 'The Java Platform consists of the Java application programming interfaces (APIs) and the Java Virtual Machine (JVM)' [27]. Java APIs refer to the libraries that are pre-compiled and could be used without having to be compiled again. Examples include the *java.io* and *java.lang* libraries. A complete Java project itself will contain programmer-readable source code written in the Java programming language along with usage of APIs or libraries to reduce the amount of work needed to be done in implementing certain algorithms. This package will be compiled into bytecodes with the command *javac MainApp.java*, creating class files which are then fed into the JVM to be further interpreted as shown in Figure 4 [4].

The JVM is the most important element in the Java Runtime Environment (JRE) where Java bytecodes are run from [28]. This JVM is entirely run by the stack frame, with each instructions contained in the compiled classes being pushed onto the stack frame when the application is being run. In order to understand further, we will have to use the Oracle's Java bytecode decompiler command *javap* in order to turn the compiled classes that are in binary files into human-readable bytecodes in terms of instructions.

To put it briefly from a different viewpoint, Figure 5 [5] allows us to determine fundamentally how any Java applications run. When the JVM is started, classes and libraries will be loaded and interpreted by the system, before it is compiled into bytecodes, executed, and cleared by the garbage collector when the application terminates.

2.5.1 Bytecodes

Understanding the instructions represented by the bytecodes is fundamental to learning how the execution of the algorithms work. Java bytecodes are similar to assembly language, though they are merely an intermediary between the source code and the JVM, with underlying machine execution occurring within the Java execution engine itself. This process is not mentioned by Oracle.

To start with dissecting a piece of class file into human-readable instructions, we will have to use the command *javap -c MainApp.class*[19]. An in-

[19]More information can be found on the official Oracle documentation site: `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`.

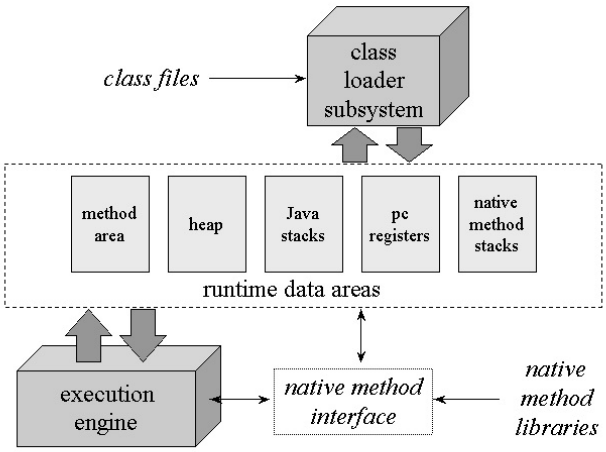


Figure 4: 'The internal architecture of the Java virtual machine.' Adapted from [4].

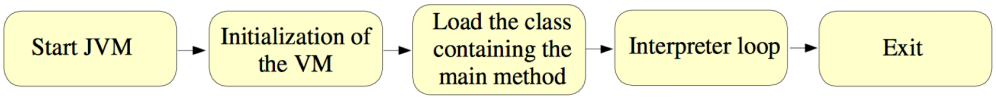


Figure 5: 'Simple view of the JVM life-cycle.' Adapted from [5].

stance of the code in Chapter 4.1.2, Listing 5 is decompiled with its instructions shown in Listing 1.

```
Compiled from "MainApp.java"
public class MainApp {
  public MainApp();
    Code:
       0: aload_0
       1: invokespecial #1                  //
          Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: ldc2_w        #2                  //
          long 1000l
       3: invokestatic  #4                  //
          Method java/lang/Thread.sleep:(J)V
       6: goto          0
       9: astore_1
      10: aload_1
      11: invokevirtual #6                  //
        Method
        java/lang/InterruptedException.printStackTrace:()V
      14: goto          0
    Exception table:
       from    to  target type
           0     6     9   Class
              java/lang/InterruptedException
}
```

Listing 1: Disassembled bytecodes for simple System.out loop.

There are various operands and opcodes listed. The opcode *aload_0* pushes the value at index 0 of the local variable table onto the stack frame. The next opcode *invokespecial* calls the constructor for *java.land.Object*, loading the library for initializing an object in Java before returning. The main method in the next part of the listing follows the same logic. The numerical before the opcode itself represents the location the opcode is stored

within the bytecode array itself and the location where the instruction is called[20].

Each and every instruction that is executed by the execution engine will use up a certain amount of power. It is with this logic that we hope to dissect the overall power consumed by the entire program into several smaller sections that are of sizes in groups of bytecodes. Through this, we hope to understand a pattern of how bytecodes correlate with the amount of total power consumed by the Java algorithm itself during the execution stage as Java has a Just-In-Time compiler that executes mostly during its runtime and thus, most of the power consumption will logically occur when the application is being run.

2.5.1.1 Power profile of JVM

Most of the dynamic power consumption a Java application has during its execution stage, is a result of dynamic loading and running of libraries, primitive values and other I/O calls. Particular of relevance to us in the paper 'An Energy Consumption Model for an Embedded Java Virtual Machine' is the breakdown of overhead costs for running any Java applications. The interpreter stage of the Java life-cycle takes up an approximate of 70% of energy usage every time the application is run [5].

Changing the arrangement of codes and data structures of variables will not lead to a reduction in these overheads - they are costs that are present even when running an empty Java application. What is more of a concern to us would be the variable costs in implementing an application, and this will be further elaborated when we have done our own testing on PowerTOP.

[20]To understand more about bytecodes, we recommend this webpage by IBM: `https://www.ibm.com/developerworks/library/it-haggar_bytecode/`.

3 Design and Implementation

In order to understand the relationship between energy complexity with different Java algorithms, an application in Java is coded. The architecture of the Java program is shown in Figure 6. In particular, other than the GUI and decompilation done using Oracle's *javap* command, the other parts that are highlighted yellow are considered back-end system of the entire program.

A brief overview of how this program work is as follow with the numbers enumerating the process on Figure 6 itself:

1. The user will prepare his own Java application, tested it that it can compile and it will run for a period of more than 30 seconds. The entire pre-compiled Java package will be prepared as the folder that will be tested.

2. The user opens the GUI application and proceed on according to the documentation provided from the help menu. Selection of energy calculation will start the entire process of power estimation.

3. On the front-end, *javap* will be run, decompiling the Java file where the main method is located at and displaying it on the GUI itself.

4. On the back-end, the compiled Java package jar file will be pushed onto Gitlab.

5. On the second machine, Jenkins CI will clone the targeted jar pushed onto Gitlab.

6. PowerTOP will be run simultaneously with the targeted Jar file. The power estimation is done over a few iterations of a few seconds long with the results being outputted into several CSV files. The results will be transferred back to the main GUI via TCP.

7. The power estimated along with the decompiled bytecodes will be stored into the database for further analysis.

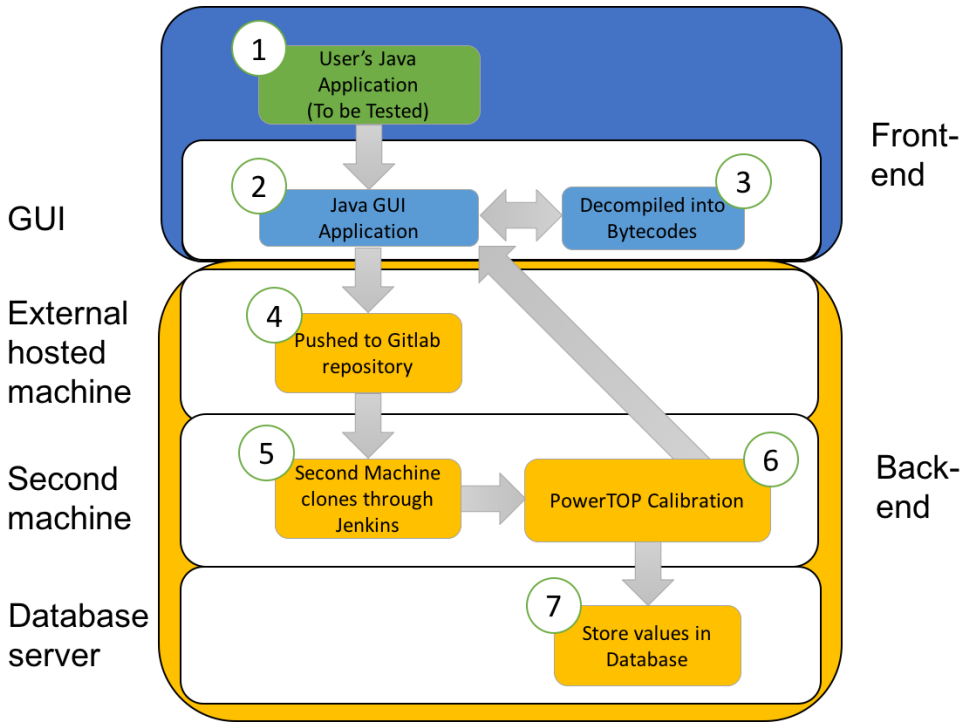


Figure 6: Architecture of the interaction between GUI and back-end system.

3.1 Graphical Interface

The interface developed to understand a relationship between power consumption and decompiled bytecodes of Java algorithms is done using a GUI. The reason why the GUI is used instead of the command line interface is to prepare for future extensions like decomposing the power consumption of the program into separate sections of repeated patterns in the decompiled bytecodes. While a command line interface tool might reduce the amount of time for development, it depends on the requirement of the tool and in this case, part of the objective is to correlate the power consumption with bytecodes decompiled using a graphical method. Hence, building the tool on command line interface will not necessarily differ in terms of time taken.

The graphical interface for this tool is developed using JavaFX 8.0[21]. This particular framework is chosen due to the lack of requirenent for an

[21]Relevant information can be found here: `https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm`.

intense UI for the developers. A basic UI will be enough for the purpose of correlating power with bytecodes, and JavaFX 8.0 being the latest framework being supported by Oracle that is well-documented on the internet will be a good framework for the GUI of the tool.

In addiction, the tool is coded with the latest Java Development Kit 8 (JDK 8) as it enables certain features like Lambda expressions, which makes coding much neater and easier for future extensions to happen. JDK 8 is also compatible with JavaFX 8.0.

3.1.1 Explanation of interface

After opening the program jar file, the power estimation selection can be started by choosing the selected option before starting the process of power measurement. Before the power measurement is done on the back-end system, the decompiled bytecodes will be shown for the main java file. This is obtained easily from the *javap* command by Oracle's JDK, which will be done using a shell script implemented[22].

A cached version of the power measurement will be shown with various data such as the minimum, maximum and average power, along with the number of data points stored within the database if a prior version of the same bytecodes is stored within the database. Otherwise, the printed values of the cached version will simply be 0. This is seen in Figure 7.

The *CACHED* values represent data points obtained from the database with the same code base detected via its bytecodes. *NEW* represents the latest data obtained during this instance of measurement. *OVERALL* shows the addition between the previous stored data with the newest data and will present themselves as *CACHED* the next time this code base is used for power measurement.

As for the type of statistics shown, there are the average, minimum, maximum and quantity of data for each of the respective fields. These statistics shown are basic values to give an estimation to the users how reliable the data are, and if there exists any huge drastic change in terms of power consumption during the current instance.

There is an option for exporting the stored data from the database if the user require further information and statistical analysis or sorting of outliers to fit a more concrete model. The data will be exported in a CSV format.

[22]More information to the shell script can be found in the Appendix B, Listings 23.

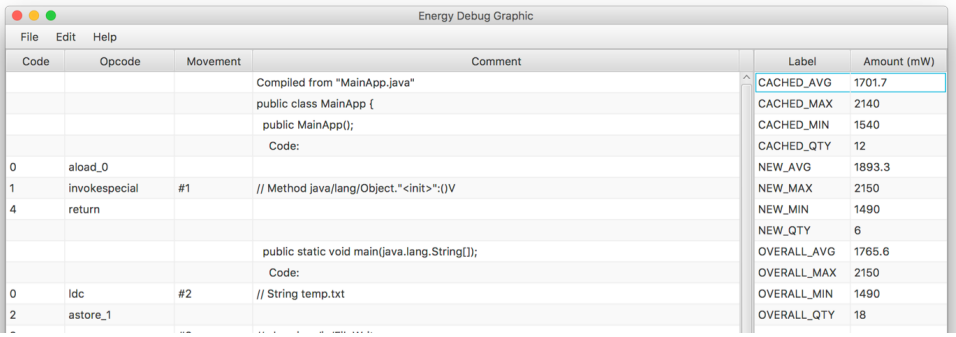


Figure 7: Graphical Interface of the Java Program after power measurement is done.

3.2 Back-end System

The main bulk of the design behind this Java program lies unseen. Other than having a database that stores the data collected for further analysis, there is the usage of a Continuous Integration (CI) tool and git for connecting between the front-end to the back-end.

3.2.1 MongoDB

MongoDB is the database system chosen for this project. The rationale behind why a NoSQL database is chosen is mainly due to the need for a flexible storage without having a set schema behind the datasets. This database is mainly used for storage of decompiled bytecodes for identification purposes, as well as future development of having recommendations between similar bytecodes that aid in improving energy efficiency of developers' algorithms. Originally, a clone detection feature is planned to be implemented via other open source frameworks for this identification, however due to the lack of suitable frameworks, this feature is not implemented.

Other than the storage of decompiled bytecodes, a list of power consumption measured on the second machine will be stored as well for programs having the same bytecodes so that they could be used for statistic analysis. This list of power could be exported for further analysis should the users need them as mentioned earlier in section 3.1.

```
{
  "_id": ObjectId("592b04e61bf9f92457b10457"),
  "decompileModels": [{
    "code": "",
    "opCode": "",
    "movement": "",
    "comment": ""
  },
  ...
  ],
  "averageEnergy": 1699.26,
  "energyList": [1930, ..., 1330]
}
```

Listing 2: Schema of the stored database values

3.2.2 Jenkins CI and Gitlab

The interconnecting link between the second machine and the UI is implemented using Jenkins and Gitlab. Due to the fact that PowerTOP could only be run on a machine running on the Linux OS, there is a need for the transfer of file to a second machine. This is currently achieved via a shell code[23] on the primary user machine. This shell code will push the required files onto a private repository on Gitlab before Jenkins CI clones the required jar file into the second machine.

The primary reason behind why Gitlab and Jenkins are used in conjoint is because of the need for automation. File transfer is easily achievable by git, and once files are pushed onto the repository, Gitlab will send a call to jenkins via known webhooks for the next part on the pipeline to start.

[23] A copy of the shell code is listed in Appendix B, listing 24.

3.2.3 Second machine

The second machine has the following specifications:

CPU	Intel Core i3-380UM 1.333 GHz
RAM	4096 MB, DDR3; PC-10700 (667 MHz)
GPU	Intel Graphics Media Accelerator (GMA) HD Graphics
OS	Lubuntu 16.04 32-bit
Battery	57 Wh Lithium-ion

Table 3: Specifications of the second machine (Lenovo Thinkpad Edge)

The reason why Lubuntu[24] is used for the second machine is because it is lightweight and has fewer additional components installed alongside the OS. This isolates the possible interferences to the power measurement that will be done using PowerTOP.

3.2.4 PowerTOP Power Measurement

PowerTOP 2.8 is the power measurement being used to calculate energy efficiency of the application. Due to the nature of the program, energy measured is in terms of Watts instead and are of a time-based unit. Calibration will have to be done to turn these values into accurate energy usage, but for the scope of this project in comparing the relative differences, it suffices for power to be used without conversion.

After acquiring the required jar file from Gitlab, the second machine will be running both PowerTOP and the targeted jar file simultaneously with power measurement taking place. Due to the fact that the official PowerTOP currently only runs as an independent utility software and that it could not be attached to the targeted jar file for a holistic power measurement targeted to the program itself[25], there will be some uncertainty in the power measured.

PowerTOP will output power measurement into CSV files for every five seconds up to a total of five iterations before sending the values back to the main machine via TCP on NetCat. All these commands are carried out via a shell code on Jenkins itself for automation[26].

[24]Lubuntu OS stands for Light Ubuntu, a lightweight distribution of the official Ubuntu. More information can be found here: `http://lubuntu.net`.

[25]This is in fact the nature of PowerTOP as it is built to estimate power usage via sleep cycles within the CPU

[26]Relevant code can be found in Appendix B, listing 25.

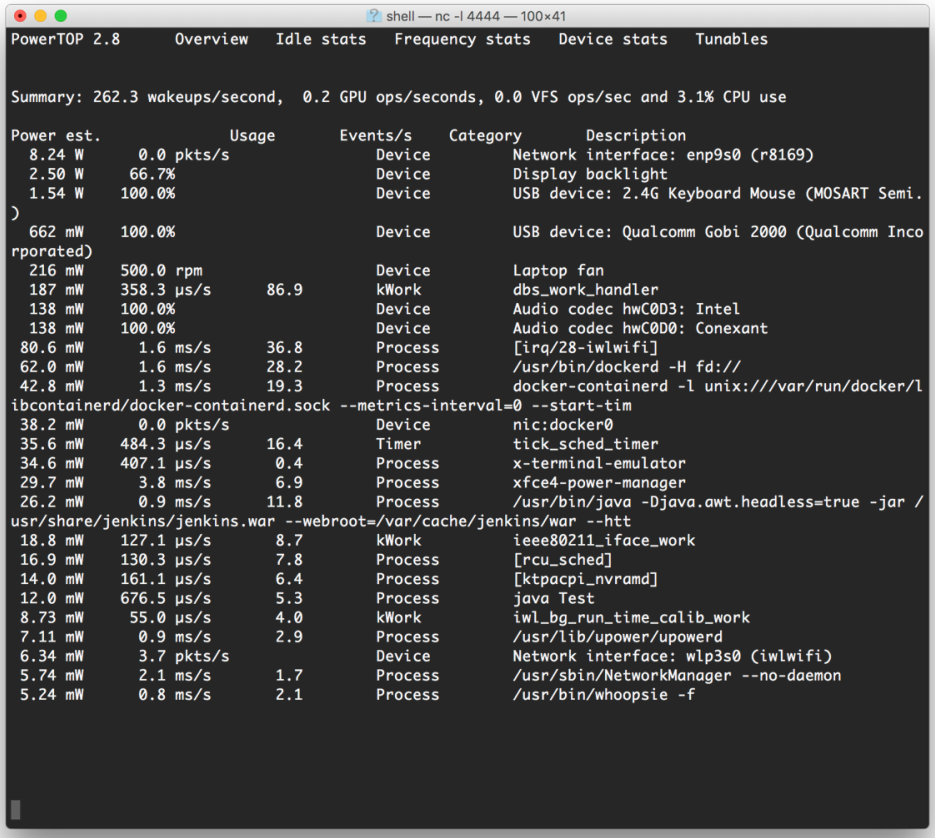


Figure 8: A simple snippet showing the working of PowerTOP after having its values copied onto terminal from the second machine.

The time interval chosen for measurement of power could be chosen. By default, it is 20 seconds. However, due to the need for a more sensitive and accurate measurement of the target application running, a shorter period is chosen. Five seconds have been chosen for the reason that it is short enough for the values of power measured to be sensitive while ensuring the entire measurement does not require too much time. This also restricts the applications that can be tested to those with at least a running period of more than 25 seconds, though a wrapper using the while loop would suffice in solving this issue.

Another limitation in terms of applications that could be tested is a result of the shell code being executed in jenkins. As seen from the jenkins shell code listed over at Appendix B, Listing 25, the user's Java target jar and PowerTOP are being called with the terminal command '&', and due to the nature of the shell command itself, all inputs have to be redirected to */dev/null*. As for the outputs, they are redirected to */dev/null* as well as previous testings show values being erratic when the target jar application prints output onto the terminal itself via *System.out*. Thus, this formally limit applications that contain command line arguments since no input are being received. Further discussions will be carried out in chapter 4 regarding these I/O read and write.

4 Experimental Evaluation

Various tests have been done using PowerTOP in estimating the amount of power usage of a Java application with varying conditions. In the first section of this chapter, different algorthms are used so as to understand more about how the Java APIs and source codes create a differnece in power usage. A pseudo-idle algorithm is first examined so as to gain a knowledge of the underlying power consumed by an 'idle' JVM. After which, simple I/Os such as the *System.in* and *System.out* APIs are tested to obtain a further baseline in terms of how much power is consumed. Different search and sorting algorithms come next as they utilize a more complex algorithm comparatively, which we are testing to see if there is any huge power consumption difference. Finally, a further subsection of I/O reads and writes are tested. This subsection differs from the simple I/Os mentioned previously as they deal more with local file systems and less with the command line interface which the *System.in* and *System.out* deal with.

In the second section, we discuss further experimentations done on the power optimization tool, PowerTOP itself to see how certain tweaks will cause a change in the power estimated. This is done so as to further understand if the results which we obtain in the previous section are reliable and accurate.

The source codes of the tests done is located at `https://github.com/lunalite/javaGreenOfficial`.

Power estimation (mW)	**Description**
8150	Network Interface: enp9s0 (r8169)
2640-2790	Display backlight
1070-1170	USB device: 2.4G Keyboard Mouse
207-297	dbs_work_handler
76.5-130	Laptop fan
4.9-40.7	java app sleeping

Table 4: Comparison between different power estimation

Table 4 shows a list of example power estimated value on the machine tested while running PowerTOP in iterations of 5 seconds. To put the values into perspective, the addition of all components of power discharge rate will total to the amount of discharge rate happening on the machine. The current second machine, according to the specifications in Chapter 3.2.3, has a 57

Wh[27] Lithium-ion battery. By default, the full energy stored within the battery will be less than the amount speicified at about 47.52 Wh[28].

Consequently, according to Table 4, the amount of discharge rate is the sum of all processes shown on the interface itself. As an example, if the discharge rate is at 14.8W , a laptop by design then will last for

$$47.52\ Wh/14.8\ W = 3.21\ hours$$

if and only if the discharge rate remains constant. Logically, if we turn off the network interface, the new discharge rate will be at 6.65 W, allowing for the laptop to last for an even longer period of time considering no other influences.

To further understand what the values listed imply, Table 5 shows various power ratings that household appliances have. Thus, a normal laptop would have a typical discharge rate of 50-60 W, or 50,000-60,000 mW. Ultimately, this discharge rate differs from machine to machine.

Power rating (W)	**Description**
25-75	Ceiling fan
50-60	Typical average for Laptop
60-100	Light bulb
175-250	Typical average for desktops
500	Washing Machine
1200-3000	Electric Kettle

Table 5: Perspective of power ratings for different household appliances [7].

In the following section, changes to Java algorithms will be made and experimented on for the purpose of understand the correlation between the algorithm itself with the estimated power rating obtained from the tool developed. A few algorithms will be compared in terms of their bytecode to sieve an understanding as to how the system interact with the JVM according to PowerTOP.

After which, the next section will be about several tests done on PowerTOP itself so as to gain a better understand of the power estimated, and how accurate it is.

[27]Wh here refers to Watts-hour.

[28]This value is obtained from the *upower* command.

4.1 Algorithms

Different algorithms are varied in this section that range from simple *System.out.println* to more advanced algorithms such as the binary search and different sorting algorithms. This is done to gain and understanding of how estimated power obtained from PowerTOP correlates with the algorithm used using the tool developed. A few algorithms will be compared in terms of their bytecodes to sieve out knowledge behind why there is a difference in power.

Most of the testing done in this section is of the comparison between different types of I/O packages such as the java.io and the java.nio package to determine the difference in terms of power estimated, as well as how the file type might cause a change in the power estimation. The results are being discussed and shown in each respective subsections.

4.1.1 Pseudo-idle JVM

The starting point of the experiment will be done to obtain the idle power consumption of the JVM. This is measured using a while loop with the main code turning the thread to sleep, or more accurately, to a 'TIMED_WAITING' state. There are various states which a thread in Java application can be. They are as follow[29]:

- Thread.State.EW - A thread that has not yet started is in this state.
- Thread.State.RUNNABLE - A thread executing in the Java virtual machine is in this state.
- Thread.State.BLOCKED - A thread that is blocked waiting for a monitor lock is in this state.
- Thread.State.WAITING - A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- Thread.State.TIMED_WAITING - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- Thread.State.TERMINATED - A thread that has exited is in this state.

[29]The list is obtained from Oracle's Java documentation at `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html`.

According to Oracle, these states are virtual states and do not reflect any operating system states. However, it remains that the VM threads will be executed by the native OS threads as well, thus, the Thread.State will be the state for the native threads and could be regarded as its virtual state. When we put the current running thread to sleep (in this case, the currentThread is the main method), the thread running the Java application will be in a TIMED_WAITING mode which does nothing. As a result, using this algorithm, we can gain a base measure of comparison with other forms of algorithms. This algorthim allows us to understand the current power overhead for a JVM.

As can be seen from Table 6, the obtained data ranges from 4.9 mW to 40.7 mW with an average of 14.0 mW. This is the base overhead for running the JVM with no calculations and I/O being done on it. This overhead is largely a result of the interpreter of JVM according to the research done by Lafond & Lilius [5]. However, the absolute value of 14.0 mW here means nothing when comparing it to any other forms of power estimation done by other researchers in view of a different machine and underlying kernel used. Thus, the significane behind this value will only be used as a comparison between other forms of estimations done on the same machine, and it will be shown in the next few subsections.

Power estimation (mW)	**Description**
4.9-40.7 (14.0)	Thread.sleep

Table 6: Power comparison obtained for Thread.sleep API.

```
while (true) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Listing 3: Java code for a sleep loop.

To further dissect what is happening in the loop, we decompile the class files with Oracle's *javap* to obtain Listing 4. In fact, the only other thing

happening is when the Java's Just-In-Time compiler calls for the java.lang API so as to change the state of the currentThread to 'TIMED_WAITING'.

```
public static void main(java.lang.String[]);
  Code:
     0: ldc2_w        #2                  // long
        1000l
     3: invokestatic  #4                  // Method
        java/lang/Thread.sleep:(J)V
     6: goto          0
     9: astore_1
    10: aload_1
    11: invokevirtual #6                  // Method
       java/lang/InterruptedException.printStackTrace:()V
    14: goto          0
  Exception table:
     from    to  target type
         0     6     9   Class
            java/lang/InterruptedException
```

Listing 4: Java bytecodes for Thread.sleep loop.

4.1.2 Simple command line I/O

4.1.2.1 System.out

In this subsection, we examine the amount of power used for a Java application printing an output onto the command line. Initially when the experiment is done with just a normal *System.out* application running, a largely varying result is obtained as shown in Table 7. It was thought that this occurred due to the outputting of text on the command line interface and indeed, when the output of the application is redirected to */dev/null*, the result obtained is more precise. This is as expected in view of the requirement for greater resource consumption for information to be transferred out of the CPU (output is shown in the case of System.out) through the system buses, taking a longer time and greater energy.

```
while (true) {
  System.out.println("H");
```

Power estimation (mW)	Description
921-1120 (1080)	System.out >/dev/null
13.1-995 (311)	System.out
50.8-1620 (728)	System.out >/dev/null (Two threads)
3.98-29.5 (10.4)	System.in
Error	System.in </dev/null

Table 7: Power comparison obtained for simple System.out and System.in APIs.

```
}
```

Listing 5: Java code for a System.out loop.

When the output of *System.out* is being redirected to */dev/null*, this Java application will in fact, not be doing any I/O calls. Instead, what is happening is more of the application calling the java.io libraries with the JVM's Just-In-Time compiler, causing a markedly greater amount of power consumed as compared to the average power, 14.0 mW, for the overhead of a Java application. This can be seen in Listing 6 where the only difference between it and the base comparison of sleep loop's bytecodes in Listing 4 is the library that is being called, and the thread that is in the 'RUNNING' state here all the time.

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #2                  // Field
        java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #3                  // String
        H
     5: invokevirtual #4                  // Method
        java/io/PrintStream.println:(Ljava/lang/String;)V
     8: goto          0
```

Listing 6: Java bytecodes for System.out loop.

As mentioned earlier, there is a great variation in terms of power estimated for *System.out* not redirected. We think that this is a result of multi-threading and have attempted in measuring the power obtained when more than one thread is being used for the *System.out >/dev/null* subject.

Indeed, the variation remains huge despite the fact that the output is redirected to */dev/null*. Further discussions to multi-threading will be held in the next section, Chapter 4.2.2.

```
class ThreadObj implements Runnable {
  private Thread t;
  private String threadName;

  ThreadObj(String name) {
    threadName = name;
  }

  public void run() {
    while (true) {
      System.out.println(threadName);
    }
  }

  public void start() {
    if (t == null) {
      t = new Thread(this, threadName);
      t.start();
    }
  }
}

public static void main(String args[]) {
  ThreadObj t1 = new ThreadObj("1");
  ThreadObj t2 = new ThreadObj("2");
  t1.start();
  t2.start();
}
```

Listing 7: Java code for a System.out multi-threading loop. Reference taken from tutorialpoint.com

4.1.2.2 System.in

For *System.in*, when no input is obtained during the power measurement, the power obtained is of a similar range to when the JVM is idling and no I/O calls happening since the entire application is waiting for a non-existent input. For the case where input is obtained, it will be tested in Chapter 4.1.4.

No power measurement could be obtained when the input is redirected from */dev/null* as EOF is being sent all the time to the *System.in* (this is the nature of the tool developed), resulting in *NoSuchElementException* and terminating the JVM prematurely.

```
while (true) {
  Scanner sc = new Scanner(System.in);
  sc.next();
}
```

Listing 8: Java code for a System.in loop.

```
public static void main(java.lang.String[]);
  Code:
     0: new           #2                  // class
        java/util/Scanner
     3: dup
     4: getstatic     #3                  // Field
        java/lang/System.in:Ljava/io/InputStream;
     7: invokespecial #4                  // Method
        java/util/Scanner."<init>":(Ljava/io/InputStream;)V
    10: astore_1
    11: aload_1
    12: invokevirtual #5                  // Method
       java/util/Scanner.next:()Ljava/lang/String;
    15: pop
    16: goto          0
```

Listing 9: Java bytecodes for System.in loop.

4.1.3 Search and sorting algorithms

Searching and sorting algorithms function differently from simple *System.out* and *System.in* I/O reads as they consist of calculations working out from just

within the CPU itself. Listings 10 and 11 consist of binary searches, with one of it using a local class implementation and the other, using the Java API. Listings 12, 13 and 14 are comparisons done between different sorting algorithms that are all implemented locally.

Power estimation (mW)	**Description**
1030-1060 (1040)	BinarySearch - Local Class
1000-1010 (1010)	BinarySearch - Java API
1010-1090 (1020)	BubbleSort
1050-1080 (1055)	MergeSort
1040-1050 (1040)	QuickSort

Table 8: Power comparison obtained for more complex search and sorting algorithms.

```
//Generate X increasing order integers.
int[] t = NumGen.gAsc(99999999);
BinarySearch binarySearch = new BinarySearch(t);
while (true) {
  binarySearch.search(ThreadLocalRandom.current().nextInt(0,99999999
     SEARCHABLE.RECURSIVE);
}
```

Listing 10: Java source code for BinarySearch using local class files.

```
int[] t = NumGen.gAsc(99999999);
while (true) {
  Arrays.binarySearch(t,
     ThreadLocalRandom.current().nextInt(0,99999999));
}
```

Listing 11: Java source code for BinarySearch using Oracle's Java API.

As mentioned earlier, a manually implemented BinarySearch algorithm is compared with the Oracle implemented version of BinarySearch to determine if there are any differences in terms of power consumption when a Java API is called compared to a local java file being compiled. Both use the same random generator in order to obtain a random integer to be searched. The final result shows that there is not much of a difference, with a slightly greater amount of power consumed for a local class implementation, showing that it

is indeed favourable to using the prepared APIs that have been tested and optmized.

```
//Generate X ranodm integers
while (true) {
  int[] t = NumGen.randG(9999);
  BubbleSort bubbleSort = new BubbleSort(t);
  bubbleSort.sort();
}
```

Listing 12: Java source code for BubbleSort.

```
//Generate X random integers
while (true) {
  int[] t = NumGen.randG(9999);
  MergeSort mergeSort = new MergeSort(t);
  mergeSort.sort();
}
```

Listing 13: Java source code for MergeSort.

```
//Generate X random integers
while(true) {
  int[] t = NumGen.randG(9999);
  QuickSort quicksort = new QuickSort(t);
  quicksort.sort();
}
```

Listing 14: Java source code for QuickSort.

The random number generator is being called within the while loop to allow for a new set of data to be generated each time. Although this might lead to additional processing being done, we are still comparing in between the different algorithms under the same condition. Both QuickSort and MergeSort have an average time complexity of $n\ lg\ n$ while BubbleSort has a time complexity of n^2. Certainly, there are many variations which we can implement such as by changing the size of array being sorted or how randomized the array is so as to determine if these conditions would cause a difference in power consumption, but there is sufficient reason for us to extrapolate that the final outcome will be of a similar range in terms of power consumption.

Since the sort and search algorithms are all calculations done without any I/O being shown (all output redirected to */dev/null*), then fundamentally,

these algorithms and *System.out* should consume a similar amount of energy as a result of minor differences in power consumed for a small program when calculations are done solely within the processor without the need for any I/O calls.

We think that similar programs where no I/O calls or other resource-extensive APIs are called will ultimately be estimated by PowerTOP to consume a power around the value of 1000 mW. We suggest the reason is because PowerTOP estimates power consumption via the switching of processes in between the Cn-states and in this case, all the aforementioned applications will be limited by a single thread running, thus, even if the algorithm is complex, the processor will be limited to a certain amount of throughput every second[30].

As a result, using any of the algorithms for search and sort will not cause a fundamentally difference in power consumed. Using the Java APIs for such purposes might even be more power-efficient.

4.1.4 I/O Reads and Writes

In order to test actual I/O reads and writes while using the java program developed, we will have to implement everything within the code itself such that it does not require any CLI arguments. We decided to employ the use of unbuffered and buffered writers and readers with the results shown in Table 10 using the java.io package. Buffered I/O is a converted stream of unbuffered I/O where the I/O streams will be acted on an intermediary known as the buffer. As for unbuffered I/O, elements are written/read one at a time. It is logical that unbuffered I/O will be less efficient in terms of throughput. The java.nio package version of buffered writer will also be compared. A summary of the differences between the different methods is shown in Table 9.

The main differences between the java.io and java.nio package is that java.io package uses a stream-based approach while java.nio package uses a block-based approach. The java.io package will read in characters/bytes one by one in a singular manner in a stream while the java.nio package reads in the values onto a buffer before any further action is from the buffer itself. The java.nio package might seem to be quite similar with the java.io.BufferedWriter and java.io.BufferedReader methods as both wraps around an incoming stream

[30]This is excluding other JVM-based background threads which spawned as a result of certain reasons, such as the garbage collection when the application terminates.

I/O type	Description
java.io.FileReader, java.io.FileWriter	Unbuffered character stream
java.io.FileInputStream, java.io.FileOutputStream	Unbuffered byte stream
java.io.BufferedReader, java.io.BufferedWriter	Buffered character stream
java.io.BufferedInputStream, java.io.BufferedOutputStream	Buffered byte stream
java.nio	Buffer-based

Table 9: Differences in the methods of I/O call in Java.

before directing it into the intermediary buffer before any further actions are carried out.

We will compare between the FileWriter that uses a character stream approach, FileOutputStream that uses a byte stream approach, the BufferedWriter and BufferedOutputStream wrapper approach and the java.nio package writer approach using the source codes written in Listings 15, 16, 17, 18 and 19.

```
String fn = "abc.txt";
while (true) {
  try {
    FileWriter fw = new FileWriter(fn);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("Hello World");
    bw.close();
  } catch(IOException e) {
    e.printStackTrace();
  }
}
```

Listing 15: Java code for a short FileWriter loop.

```
String fn = "abc.txt";
String msg = "Hello World";
byte[] bArr = msg.getBytes();
while (true) {
```

```
  try {
    FileOutputStream fos = new FileOutputStream(fn);
    fos.write(bArr);
    fos.close();
  } catch(IOException e) {
    e.printStackTrace();
  }
}
```

Listing 16: Java code for a short FileOutputStream loop.

```
String fn = "abc.txt";
while (true) {
  try {
    FileWriter fw = new FileWriter(fn);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("Hello World");
    bw.close();
  } catch(IOException e) {
    e.printStackTrace();
  }
}
```

Listing 17: Java code for a short BufferedWriter loop.

```
String fn = "abc.txt";
String msg = "Hello World";
byte[] bArr = msg.getBytes();
while (true) {
  try {
    FileOutputStream fos = new FileOutputStream(fn);
    BufferedOutputStream bfos = new
       BufferedOutputStream(fos);
    bfos.write(bArr);
    bfos.close();
  } catch(IOException e) {
    e.printStackTrace();
  }
}
```

Listing 18: Java code for a short BufferedOutputStream loop.

```
String fn = "abc.txt";
String msg = "Hello World";
byte[] bArr = msg.getBytes();
while (true) {
  try {
    FileOutputStream fout = new
       FileOutputStream(fn);
    FileChannel fc = fout.getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    for (int i = 0; i < bArr.length; ++i) {
      buffer.put(bArr[i]);
    }
    buffer.flip();
    fc.write(buffer);
    fc.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Listing 19: Java code for a short java.NIO buffer writer.

Power estimation (mW)	Description
1190-3330 (2121.2)	Short FileWriter
1340-3370 (2538.6)	Short FileOutputStream
1200-1840 (1392.2)	Short BufferedWriter
1200-3560 (2309.3)	Short BufferedOutputStream
1490-3320 (2530)	Short java.nio buffered writer
890-1510	Short BufferedReaderA
1240-3010	Short BufferedReaderB (random)
1130-3100	Short BufferedReaderC

Table 10: Power comparison for I/O reads and writes.

As we can see from Table 10, the average power estimated for BufferedWriter is the least as compared to the other methods of stream and java.nio buffer-based solutions for writing of short strings. The experiments done excluded the instantiation of variables and conversion of these strings into bytes for byte-based solutions like FileOutputStream, BufferedOutputStream and java.nio buffered writer, but such conversion for a short string is negligible compared to the writing of bytes onto a file. The BufferedOutputStream approach and java.nio buffered writer approach consume similar power as both used a byte-based buffer approach in writing the string into a file. FileWriter itself as a character stream approach consumes on average, a lower power as compared to the byte-based approach of FileOutputStream. This is similar for BufferedWriter with BufferedOutputStream where the first uses a character stream approach while the latter uses a byte stream approach.

In order to find out the reason why there is a disparity in the power consumption, we decided to compare their disassembled bytecodes. It is however, seen that both approaches differ solely only in the Java API implementation of character- and byte-based approach with the general structure of the instructions being similar. This implies the difference in power consumption is a result from the underlying mechanism behind these APIs. One possible reason might be the need to convert the primitive byte data into characters when writing. Another reason could be due to caching being done within the CPU itself.

Three different types of buffered reader algorithms are implemented. Type A reads the same file with the same bytes encoded within without any changes. The result is a much smaller power consumption estimated as

compared to the measured power for a buffered writer. We think the reason behind this is a result of caching that led to a smaller amount of power consumption. As a result, type B and C algorithm were implemented.

```
String fn = "abc.txt";
String line = "";
try {
  FileWriter fw = new FileWriter(fn);
  BufferedWriter bw = new BufferedWriter(fw);
  bw.write("Hello World");
  bw.close();
  while (true) {
    FileReader fr = new FileReader(fn);
    BufferedReader br = new BufferedReader(fr);
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  }
} catch(IOException e) {
  e.printStackTrace();
}
```

Listing 20: Java code for a BufferedReader loop type A.

Both Type B and C algorithms implement the buffered writer and buffered reader within the loop. Due to the fact that buffered writer is implemented, causing the file to be rewritten within the loop, the resulting power consumption will be closer to the estimated power consumption for the buffered writer. However, the average power value will be lower than that of the buffered writer due to the presence of buffered reader that takes a much lower value.

Type B and C differs in the implementation of file string written into the file. The rationale behind this method is to ensure randomness and prevent caching from taking place. The eventual result is negligible, though for the algorithm that uses a fixed string, it is estimated to consume a lower power, showing that caching is indeed present.

```
String fn = "abc.txt";
String line = "";
RandomString r = new RandomString(10);
try {
  while (true) {
    FileWriter fw = new FileWriter(fn);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write(r.nextString());
    bw.close();
    FileReader fr = new FileReader(fn);
    BufferedReader br = new BufferedReader(fr);
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  }
} catch (IOException e) {
    e.printStackTrace();
}
```

Listing 21: Java code for a BufferedReader loop type B.

```
String fn = "abc.txt";
String line = "";
try {
  while (true) {
    FileWriter fw = new FileWriter(fn);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("Hello World");
    bw.close();
    FileReader fr = new FileReader(fn);
    BufferedReader br = new BufferedReader(fr);
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  }
} catch(IOException e) {
  e.printStackTrace();
```

```
}
```

Listing 22: Java code for a BufferedReader loop type C.

4.2 System and Other Processes

With the working principle of PowerTOP, we believe that other processes might affect the power estimation of the running Java application. We list various steps undertaken to determine if the effects of certain measures taken would influence the original power estimated in the targeted Java program or not. To this, we use the *System.out* loop as the benchmark for testing with the output redirected to */dev/null*. The relevant code can be found in Listing 5.

4.2.1 Duplicating the Benchmark

We tested out by duplicating the benchmark application itself to determine what PowerTOP will return. By running a second similar application with the same name, the power estimated has been doubled. When a third application is called, the power estimated on PowerTOP has tripled from the original value.

We then tested the same application by changing the file name and consequently the class name. The result is that now, a different row has been created on PowerTOP, showing that PowerTOP categorizes application based on the file and class name of a Java application. This remains so even when the command of calling the java application changes such that we do not redirect the output to */dev/null*.

4.2.2 Multi-threading

PowerTOP is a tool that governs its power estimation using Cn-states of the threads. There is sufficient reason for us to believe that a multi-threaded program will affect the power estimated, reducing the probability of confidence we have on the values obtained .

Taking reference from Table 7, the power estimated for *System.out* loop that is not redirected to */dev/null* and *System.out* loop with multi-threading shows a great disparity in its minimum and maximum values as compared to that of *System.out* loop redirected to */dev/null*. We believe that the

similarity between those two aforementioned disparity is a result of multi-threading.

The *System.out* loop without redirection will cause other threads in the CPU to start changing states as the kernel requires to display the output on the CLI. On PowerTOP, when the *System.out* loop is run, processes like `flush_to_ldisc`, `kworkeru8:0`, `kworkeru8:1`, `kworkeru8:2` and `x-terminal-emulator` immediately increased in their estimated power consumption. Due to the change in Cn-states and the limited amount of threads which the kernel runs, the states of java *System.out* loop program will be affected depending on the states of other processes, causing the huge range in power consumption estimated printed by PowerTOP.

As for the case when System.out loop is run with two threads and with the output redirected to */dev/null*, the power estimated varies greatly as well. The reason is as per stated, a result of competing threads that made the C- and P- states of the program running greatly imprecise.

****

Try testing without power plug

5 Project Evaluation

5.1 Strengths

This project is based entirely on the power estimation tool developed by Intel, PowerTOP. The experimentation results obtained and the arguments put forth are based on the premise of PowerTOP itself being a viable tool for estimating power consumption of a Java application without major influences from other areas such as caching and blocking threads. There are still certain areas which have been done well.

5.1.1 Understanding of PowerTOP

Experimentations have to be done on PowerTOP itself as a tool for measuring power. It suffices to say that several key notions have been found about PowerTOP that have to be avoided to prevent a misrepresentation of power estimation:

- Java processes with the same name running will constitute as a single process on PowerTOP itself. It is vital that this mistake is not carried out, otherwise the values obtained might be inaccurate.
- Command line I/O calls by the Java application results in a huge variation in the power estimated by the PowerTOP due to multi-threading.

5.2 Limitations

There are various limitations to the idea put forth by this project and program developed.

5.2.1 PowerTOP as the main Estimation Tool

The biggest component within the entire program is the power measurement tool that is used – PowerTOP. Although mentioned earlier that PowerTOP estimates power via the C-states and P-states of the CPU, there are various drawbacks to depending solely on PowerTOP to gather information on power consumption of various algorithms.

Faria et al. [29] argues that PowerTOP could only provide a 'superficial result' as the tool is governed 'by the improvements that this external control

is able to practice'. Alexandre et al. spoke of how PowerTOP is meant as a 'functioning tool' that optimizes energy efficiency on desktop machines.

In reply to that, even though the power estimation obtained from PowerTOP might be superficial, the requirement of the program in this project is not that strict in terms of its accuracy. That is also why such a huge range of power consumption estimated has been obtained as seen in Table 11. More importantly, the main objective of this project aims to broaden the knowledge of developers in their usage of algorithms especially in I/O calls so that they will be able to determine if a package is extremely energy consuming and if there is a better alternative for them to use. The final decision will still be taken by the developer himself if he should trust the experiments done on this project. Still, further statistical analysis could be done on the data collected regarding estimated power for the different algorithms to determine the probability of confidence.

Ultimately, we agree with what Alexandre et al. put forth in terms of how the result is superficial. There are instances where other processes in the machine (processes that could not be deactivated) affect the fundamental measuring principle behind PowerTOP as shown in Chapter 4.2, reducing the effectiveness of PowerTOP as a power estimation tool. Furthermore, since PowerTOP is based entirely on sleep and awake states of the processes, having a third-party thread that is holding and blocking other threads from being executed would dramatically change the final outcome of power estimated. Even though [29] suggested that PowerScope is currently the most efficient power measuring tool, the fact remains that it requires hardware and that is contrary to the objectives of this project.

5.2.2 Clone detection tool

A viable clone detection will aid the project tremendously especially when it could gather insight into similar bytecode patterns based on how similar they are. This function is important towards providing recommendations to developers' algorithms in finding a better alternative that uses less power. This is however, not present in this project. Currently, the bytecodes stored within the database are matched using a textual comparison, limiting any form of recommendations since this method of comparison does not take into account the patterns of decompiled bytecodes.

As this project focuses more on the study of how bytecodes and Java algorithms correlate with power consumption, clone detection is considered

a minor part that will aid in the project, but not an essential tool that is required. Although there exists a viable clone detection tool, SeByte, the webpage hosting the source code of this particular tool is not available online during this point of writing. Hence, the clone detection tool will be made as a possible future extension instead.

5.2.3 Underlying kernel

A huge limitation in this project is certainly on the kernel itself. Many other papers focus mainly on mobile phones as there are more research done into the different influences of OS with the application and the power consumption. The field of research into power estimation of a desktop is advanced as well, just that there are many different underlying mechanisms in how the operating system work, creating a variable influx in terms of how these mechanisms might affect the measurement of the PowerTOP tool.

Possible mechanisms that would affect the power measurement done by PowerTOP is listed as follow:

- Type and method of caching which will affect how I/O calls are being carried out, affecting the experiments done and the values obtained as it might differ from an actual scenario.
- Number of processors which will change the number of threads that can be run optimally and in turn, affect how PowerTOP returns estimations of power as it is based on the sleep and awake states.
- The operating system itself will determine how the kernel function in terms of dealing with resources and deadlocks. This affect the states of other running threads, especially those of the Java processes that are being run and the values obtained.

5.2.4 Implementation

5 seconds iterations with a while loop might result in more loops being run for a certian algorithm as compared to another algorithm.

6 Conclusion

We summarize the result of power estimation done on various algorithms in Chapter 4 over in Table 11. There are several power estimations done using PowerTOP using different algorithms.

We agree that multi-threading will influence the measurement of power itself using PowerTOP with a huge variation in power estimation for *System.out* loop without redirection and the *System.out* loop with two threads running simultaneously, making the measurement and conclusion inaccurate for these sets of data. We also agree that usually, a single thread will be used for simple algorithms and searching or sorting algorithms, causing no huge differences in terms of power estimated for the algorithm since it is limited by the thread being run.

Any algorithms involving I/O calls is bound to consume a huge amount of power compared to normal calculation-type algorithms that solely uses the processor. The comparison done between different forms of I/O calls implemented in the Java language while in a loop of writing short strings onto a file shows that java.io.BufferedWriter consumes the least amount of power according to PowerTOP as compared to other forms of I/O writes. We have also tried finding out if caching of short strings will create a huge difference in power consumption of I/O reads, with the outcome being that the difference is minor for it to be ignored.

6.1 Future Work

Ultimately, there are tonnes of experimentations that could still be conducted, especially when I/O calls are tested for long strings. There are also many possible extensions to this project that have yet to be implemented. Primary to those is that of having a clone detection tool functioning simultaneously on the back-end database so as for recommendations purposes. One of the prime reasons why developers do not truly re-write their algorithms for better efficiency energy-wise is due to ignorance, and even if they know that their algorithm might be heavy in terms of energy usage, they might not be aware of what marks as a better efficiency algorithm which they could use. This is where the clone detection tool comes in as it basically scours for well-written codes without any huge hassle of having to manually search for them.

Power estimation (mW)	Description
4.9-40.7	Thread.sleep
921-1090	System.out >/dev/null
13.1-995	System.out
82.7-1620	System.out >/dev/null (Two threads)
3.98-29.5	System.in
Error	System.in </dev/null
724-1160	BinarySearch
603-1200	BubbleSort
1190-3330 (2121.2)	Short FileWriter
1340-3370 (2538.6)	Short FileOutputStream
1200-1840 (1392.2)	Short BufferedWriter
1200-3560 (2309.3)	Short BufferedOutputStream
1490-3320 (2530)	Short java.nio buffered writer
890-1510	Short BufferedReaderA
1240-3010	Short BufferedReaderB (random)
1130-3100	Short BufferedReaderC

Table 11: Summary of power estimation for various algorithms.

Another extension this program would have is as discussed, to link it to a plugin for an IDE. Nowadays, almost every single developers are using IDEs for their programming work and having to manually measure the energy consumption by using a third-parry standalone program would steer them away from it. An API is a possible add-on to this as well.

Also, as originally planned, having the energy measurement of the entire Java program break down into small bytecode-sized section will allow for further studies to be done into finding out the best possible energy-efficient and time-efficient algorithm that would make a difference to the entire program by finding the right balance.

References

[1] J. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011. doi: 10.1109/MAHC.2010.28. ID: 1.

[2] I. Keivanloo, C. K. Roy, and J. Rilling. Sebyte: A semantic clone detection tool for intermediate languages. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 247–249, 2012. ISBN 1092-8138. doi: 10.1109/ICPC.2012.6240495. ID: 1.

[3] M. Jones. Anatomy of the linux kernel, 6 June, 2007 2007. URL `https://www.ibm.com/developerworks/library/l-linux-kernel/`. [Last accessed 26 January 2017].

[4] Bill Venners. Inside the java virtual machine. URL `http://www.artima.com/insidejvm/ed2/jvm2.html`. [Last accessed 19 January 2017].

[5] Sébastien Lafond and Johan Lilius. *An Energy Consumption Model for an Embedded Java Virtual Machine*, pages 311–325. Architecture of Computing Systems - ARCS 2006: 19th International Conference, Frankfurt/Main, Germany, March 13-16, 2006. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-32766-0. doi: 10.1007/11682127_22. URL `http://dx.doi.org/10.1007/11682127_22`.

[6] R. Damaševičius, V. Štuikys, and J. Toldinas. Methods for measurement of energy consumption in mobile devices. *Metrology and Measurement Systems*, 20(3):419–430, 2013. doi: 10.2478/mms-2013-0036.

[7] DaftLogic. List of the power consumption of typical household appliances. URL `https://www.daftlogic.com/information-appliance-power-consumption.htm`. [Last accessed 3 June 2017].

[8] Internet Live Stats. Number of internet users. URL `http://www.internetlivestats.com/internet-users/`. [Last accessed 31 May 2017].

[9] E.ON. How much does it cost to keep your computer online? (lots, it turns out), 27 April 2017 . URL http://www.telegraph.co.uk/business/energy-efficiency/cost-keeping-computer-online/. [Last accessed 31 May 2017].

[10] Open University. Future energy demand and supply. URL http://www.open.edu/openlearn/ocw/mod/oucontent/view.php?printable=1&id=2405. [Last accessed 28 January 2017].

[11] International Energy Agency. Technical Report 978-92-64-26494-6, International Energy Agency, 2016. URL https://www.iea.org/newsroom/news/2016/november/world-energy-outlook-2016.html. [Last accessed 28 January 2017].

[12] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016. doi: 10.1109/MS.2015.83. ID: 1.

[13] Erik A. Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom, and Rob van Vliet. Software energy profiling: Comparing releases of a software product. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 523–532, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4205-6. doi: 10.1145/2889160.2889216. URL http://doi.acm.org/10.1145/2889160.2889216.

[14] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168841. URL http://doi.acm.org/10.1145/2168836.2168841.

[15] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, GREENS '12, pages 1–7, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1832-7. URL http://dl.acm.org/citation.cfm?id=2663779.2663780.

[16] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: A software-based tool for estimating the energy profile of android applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 3–6, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.18. URL `https://doi.org/10.1109/ICSE-C.2017.18`.

[17] TIOBE. Tiobe index for december 2016 - december headline: What is happening to good old language c?, 2016. URL `http://www.tiobe.com/tiobe-index/`. [Last accessed 29 December 2016].

[18] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin. Profiling software for energy consumption. In *2012 IEEE International Conference on Green Computing and Communications*, pages 515–522, 2012. doi: 10.1109/GreenCom.2012.86. ID: 1.

[19] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 421–424, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321699. URL `http://doi.acm.org/10.1145/1321631.1321699`.

[20] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597085. URL `http://doi.acm.org/10.1145/2597073.2597085`.

[21] Benjamin Westfield and Anandha Gopalan. Orka: A new technique to profile the energy usage of android applications. In *Proceedings of the 5th International Conference on Smart Cities and Green ICT Systems - Volume 1: SMARTGREENS*, pages 213–224, 2016. ISBN 978-989-758-184-7. doi: 10.5220/0005812202130224.

[22] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using

system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966460. URL http://doi.acm.org/10.1145/1966445.1966460.

[23] M. Minea P. Bulychev. Duplicate code detection using anti-unification. *Spring Young Researchers Colloquium on Software Engineering*, 2(2008): 51–54, 2008. doi: 10.15514/SYRCOSE-2008-2-22. URL http://syrcose.ispras.ru/2008/files/22_paper.pdf.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007. doi: 10.1109/TSE.2007.70725. URL http://ieeexplore.ieee.org/document/4288192/?part=1. ID: 1.

[25] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 36–42, 2012. doi: 10.1109/IWSC.2012.6227864. ID: 1.

[26] Aqeel Mahesri and Vibhore Vardhan. *Power Consumption Breakdown on a Modern Laptop*, pages 165–180. Power-Aware Computer Systems: 4th International Workshop, PACS 2004, Portland, OR, USA, December 5, 2004, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31485-1. doi: 10.1007/11574859_12. URL http://dx.doi.org/10.1007/11574859_12.

[27] Oracle. Essentials, part 1, lesson 1: Compiling & running a simple program. URL http://www.oracle.com/technetwork/java/compile-136656.html. [Last accessed 4 June 2017].

[28] E. Sagynov. Understanding jvm internals, from basic structure to java se 7 features. URL http://www.tiobe.com/tiobe-index/. [Last accessed 4 June 2017].

[29] A. W. C. Faria, L. P. d. Aguiar, D. D. Lara, and A. A. F. Loureiro. Comparative analysis of power consumption in the implementation of arithmetic algorithms. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1247–1254, 2011. ISBN 2324-898X. doi: 10.1109/TrustCom.2011.256. ID: 1.

Appendix A Definition of terms

In this paper, there are various terms used that might be interchangeable.

Desktop
: Desktops are machines that are not portable, and are frequently placed on a workstation and used there. Typically, desktops are much faster and consume more energy as compared to laptops, the portable versions of desktops.

Energy
: This term refers fundamentally to the amount of electrical energy being consumed when the machine is functional. In this paper, it is referred to interchangeably with 'power'.

Machine
: Machine refers to any digital systems in this paper, be it a laptop, a mobile phone, or any desktops.

Personal Computers
: Personal Computers refer to either desktops or laptops that are usually used for work or other purposes, and are bigger than any handheld devices such as mobile phones.

Power
: Power refers to the amount of energy being used with focus on the energy consumption per unit time. The SI unit of Power is Watts, and in this paper, it holds a similar meaning to that of 'energy'.

Appendix B Source Code Snippets of Java Program

```
#!/bin/bash
#####################################################
#
#This bash files makes compiling, jar and
   decompiling of the java package easier while
   maintaining them in a neat single file location.
#
#####################################################

#This compiles the required java files.
cd $FOLDER_INPUT
if [ ! -d "./build" ]; then
  mkdir build
else
  rm -rf ./build/*
fi
javac -d ./build ${MAIN_INPUT}

#This creates the manifest for preparation of jar
if [ ! -d "./build/META-INF" ]; then
  mkdir ./build/META-INF
fi

#This creates the file MANIFEST.MF
MAIN_INPUT_JAVA=${MAIN_INPUT//${FOLDER_INPUT}}
MAIN_INPUT_JAVA=${MAIN_INPUT_JAVA//.java}
MAIN_INPUT_JAVA=${MAIN_INPUT_JAVA/\/}
echo Main-class: ${MAIN_INPUT_JAVA} >
   build/META-INF/MANIFEST.MF

#This section jars the FOLDER_INPUT
cd build
jar cmvf META-INF/MANIFEST.MF target.jar *
```

```
#And obtain the decompilation of the main file
echo Obtaining decompilation...
javap -c ${MAIN_INPUT_JAVA}.class > decompiledMain
cat decompiledMain
```

Listing 23: Java decompile shell code

```
#!/bin/sh

#####################################################
#
#gitlab.sh
#
#This file aids in git pushing to a different
#branch so as to ensure the build folder is
#always pushed.
#
#####################################################

cd $FOLDER_INPUT

if [ -d .git ]; then
  echo .git;
else
  git init
  git remote add origin
     git@gitlab.doc.ic.ac.uk:hdk216/javaGreen.git
fi;


git rm --cached -r .

#Add in a file to obtain current machine's IPADDR
ifconfig en0 | grep inet | grep -v inet6 | awk
   '{print $2}' > ipadd

#Checking out to new branch and push
git checkout -b epoch$(date +%s)
```

```
#Delete all except current branch
git branch -D `git branch | grep -E epoch.*`

git add build/
git add ipadd
git commit -m 'no comments'
git push

#Retrieving ref names
LOCAL_BRANCH=`git for-each-ref
   --format='%(refname)' | egrep 'epoch' | awk -F/
   'NR==1{print $NF}'`

git branch -r | egrep ^..origin/epoch.*$ | egrep -v
   ${LOCAL_BRANCH} | while read line; do
  if [ -n '${line}' ]; then
    branch=`echo ${line} | awk -F/ '{print $NF}'`
    git push origin :${branch}
  else
    echo 'Remote branch not found'
  fi
done
```

Listing 24: Gitlab export shell code

```
PATH=$PATH:/usr/sbin
counter=0

cd `find . -type d -name build`
cd ..

/home/hdk216/Documents/tpt.sh &
java -jar ./build/target.jar < /dev/null >
   /dev/null &
p=$!
while kill -0 $p
do
  counter=$((counter+1))
  if [ "$counter" -ge 5 ]; then
    break
  fi
  sleep 5
done

cat powertop* | grep target.jar > a
nc $(cat ipadd) 4444 < a
```

Listing 25: jenkins shell code