FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Ranking Compiler Versions by Energy Efficiency

Pedro Miguel Oliveira Azevedo

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Paulo de Sousa Ferreira Fernandes

Second Supervisor: Prof. João Saraiva

July 26, 2023

Ranking Compiler Versions by Energy Efficiency

Pedro Miguel Oliveira Azevedo

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. João Carlos Viegas Martins Bispo
Referee: Prof. João Paulo de Sousa Ferreira Fernandes
Referee: Prof. Wellington de Oliveira

July 26, 2023

Resumo

Devido ao aumento dos preços da energia e a uma sociedade cada vez mais consciente do ambiente, tem havido um esforço para reduzir o consumo de energia em todos os aspectos da vida, mesmo no desenvolvimento de software. Um desses esforços é melhorar a eficiência energética nos centros de dados, sistemas cujo consumo total de energia deverá atingir 4,5% de todo o consumo de energia a nível mundial até 2025 [33]. Este enorme consumo de energia deve-se à quantidade de dados processados diariamente e os programadores de software têm de encontrar formas de o reduzir nestes sistemas e no seu desenvolvimento diário.

Uma área com potencial para melhorar o consumo de energia dos programadores é a forma como um compilador/interpretador influencia o consumo de energia dos programas que produz. Dada a sua importância na vida quotidiana dos programadores e no desenvolvimento dos programas, seria de esperar que fossem feitos alguns esforços para otimizar o desempenho energético dos compiladores/intérpretes. No entanto, será que isso é verdade? Será que as linguagens de programação são construídas a pensar na eficiência energética e melhoram a sua eficiência energética a cada atualização? Esta dissertação pretende responder a esta questão avaliando a progressão do consumo de energia em várias versões do mesmo compilador/interpretador para um determinado conjunto de linguagens de programação, C, C++, Java e Python, utilizando dez programas de teste do projeto Computer Language Benchmark Game (CLBG).

A eficiência energética das linguagens de programação não é um novo objeto de estudo, mas a forma como o consumo de energia evoluiu em várias versões do mesmo compilador/interpretador, sim. Além disso, a informação mais próxima que recolhi sobre este tema estava relacionada com a evolução das características de uma linguagem de programação [30] e a comparação do consumo de energia entre vários compiladores C++ [26].

Para colmatar a ausência de informação relacionada com a versão do compilador, a comparação entre eles é o principal objetivo desta dissertação. Também fornecerá mais informações sobre a evolução do consumo de energia dos referidos compiladores.

Para que qualquer conclusão sobre a evolução dos compiladores/interpretadores possa ser verificada, esta dissertação define uma metodologia para a seleção das linguagens de programação, dos compiladores e versões para essas linguagens, dos programas utilizados para os testes, das ferramentas utilizadas para a recolha e tratamento dos dados e dos testes estatísticos utilizados para esse tratamento.

A partir dos dados recolhidos das experiências efectuadas, foi possível verificar as versões mais eficientes em termos energéticos para cada uma das linguagens estudadas, através de um sistema de classificação baseado no desempenho de cada versão nos programas de teste do CLBG. Encontrámos também uma tendência discreta para a diminuição do consumo de energia. No entanto, há demasiada oscilação nestes resultados dos programas para confirmar que os programadores de compiladores/intérpretes consideram a eficiência energética quando desenvolvem actualizações e o performance das versões após a melhor aparentam deteriorar.

Abstract

Due to the increasing energy prices and an increasingly environmentally conscious society, there has been an effort to reduce energy consumption in all aspects of life, even in software development. One such effort is to improve energy efficiency in Data Centers, systems whose total energy consumption is expected to reach 4.5% of all energy consumption worldwide by 2025 [33]. This massive energy consumption is due to the amount of data processed daily, and software developers must find ways to reduce it in these power-hungry systems and their daily development.

One area with the potential to improve developers' energy consumption is how a compiler/interpreter influences the energy consumption of the programs it produces. Given the importance of it in developers' daily lives and programs' development, we would expect some efforts to be made in optimizing the energy performance of compilers/interpreters. However, is this true? Are programming languages built with energy efficiency in mind and improving their energy efficiency in each update? This dissertation aims to answer this question by evaluating the progression of energy consumption across multiple versions of the same compiler/interpreter for a given set of programming languages, C, C++, Java, and Python, using ten benchmark programs from the Computer Language Benchmark Game(CLBG) project.

Energy efficiency across programming languages is not a new subject of study, but how energy consumption evolved across multiple versions of the same compiler/interpreter is. In addition, the closest information gathered on this topic was related to the evolution of features in a programming language [30] and the comparison of energy consumption between multiple C++ compilers [26].

In order to fill the absence of information related to the compiler version, the comparison between them is the main focus of this dissertation. It will also provide further insight into the evolution of energy consumption of said compilers.

So that any conclusion on the evolution of compilers/interpreters reached can be verified, this dissertation defines a methodology used to select the programming languages, the compilers and versions for said languages, the programs used for testing, and the tools used to gather and process data as well as statistical tests used for said processing.

From the gathered data of the experiments performed, we could verify the most energy-efficient versions for each of the studied languages using a ranking system based on the performance of each version on the test programs of CLBG. We also found a discrete trend for energy consumption to decrease. However, there is too much oscillation in these results of the programs to confirm that compiler/interpreter developers consider energy efficiency when developing updates and that the performance after the best versions seems to deteriorate.

Agradecimentos

I would like to thank my mom, and grandparents for helping me through my tough times and for helping me become the man I am today. I would also like to extend my thanks to my supervisors Prof. João Paulo Fernandes and Prof. João Saraiva for guiding me through this process of completing my Dissertation and achieving my Masters's Degree.

Pedro Miguel Oliveira Azevedo

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Contents

1 Introduction **1**
1.1 Introduction . . . 1
1.2 Dissertation Structure . . . 2

2 State of the Art **3**

3 Methodology **6**
3.1 Programming Languages . . . 6
3.2 Compiler Versions . . . 7
3.3 Test Programs . . . 7
3.4 Tools and Collected Data . . . 10
3.5 Test Framework . . . 11
3.6 Data Aggregation and Cleaning . . . 13

4 Results **15**
4.1 Presentation of Results . . . 15
4.1.1 Java Results . . . 15
4.1.2 C Results . . . 20
4.1.3 C++ Results . . . 24
4.1.4 Python Results . . . 28
4.2 Discussion of Results . . . 33

5 Conclusion **36**
5.1 Acknowledgments . . . 37

References **38**

List of Figures

3.1 Java Compilation and Execution . . . 9
3.2 C Compilation and Execution . . . 9
3.3 C++ Compilation and Execution . . . 10
3.4 Python Compilation and Execution . . . 10

4.1 Java Results Binary Trees . . . 16
4.2 Java Peak RSS Performance Binary Trees . . . 16
4.3 Java Results Fannkuch Redux . . . 17
4.4 Java Peak RSS Performance Fannkuch Redux . . . 17
4.5 Java Results Fasta . . . 18
4.6 Java Peak RSS Performance Fasta . . . 18
4.7 Java General Rank . . . 19
4.8 Java Peak RSS Rank . . . 19
4.9 C Results Binary Trees . . . 20
4.10 C Peak RSS Performance Binary Trees . . . 20
4.11 C Results Fannkuch Redux . . . 21
4.12 C Peak RSS Performance Fannkuch Redux . . . 21
4.13 C Results Fasta . . . 22
4.14 C Peak RSS Performance Fasta . . . 22
4.15 C General Rank . . . 23
4.16 C Peak RSS Rank . . . 23
4.17 C++ Results Binary Trees . . . 24
4.18 C++ Peak RSS Performance Binary Trees . . . 24
4.19 C++ Results Fannkuch Redux . . . 25
4.20 C++ Peak RSS Performance Fannkuch Redux . . . 25
4.21 C++ Results Fasta . . . 26
4.22 C++ Peak RSS Performance Fasta . . . 26
4.23 C++ General Rank . . . 27
4.24 C++ Peak RSS Rank . . . 27
4.25 Python Results Binary Trees . . . 28
4.26 Python Peak RSS Performance Binary Trees . . . 28
4.27 Python Results Fannkuch Redux . . . 29
4.28 Python Peak RSS Performance Fannkuch Redux . . . 29
4.29 Python Results Fasta . . . 30
4.30 Python Peak RSS Performance Fasta . . . 30
4.31 Python General Rank . . . 31
4.32 Python Peak RSS Rank . . . 31

List of Tables

3.1 Compiler Versions 7
3.2 Test Programs 8
3.3 Program Versions 9
3.4 C N-Body Single-Thread vs Multi-Thread 12
3.5 Services kept alive and reasoning 13
3.6 Benchmarks used for ranking by language 14

4.1 Best version of each language in terms of energy efficiency and runtime 34
4.2 Best version of each language in terms of peak energy usage 35

Abreviaturas e Símbolos

CLBG	Computer Language Benchmark Game
RAPL	Running Average Power Limit
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
GCC	GNU Compiler Collection (C/C++ Compiler)
CLANG	C/C++ Compiler
RSS	Resident Set Size

Chapter 1

Introduction

1.1 Introduction

Recently, due to the increasing energy prices and an increasingly environmentally conscious society, there has been an effort to reduce energy consumption in all aspects of life, even in software development. One such effort is improving energy efficiency in Data Centers[42] [29] [28], that in 2017 were responsible for 3% of all energy consumption and, by 2025, are expected to hit 4.5% of global energy consumption[33]. However, to improve even further optimization efforts for data centers, they should also consider the program generated by the compiler/interpreter, as the resulting programs perform various tasks such as communication and storage. In addition, the resulting programs could consume more energy than intended due to their compiler/interpreter optimization, and it is necessary to understand the full effects of these optimizations on energy consumption not only for reducing the energy consumption of data centers but also all other programs.

For any developer, the mentioned Compilers/Interpreters are essential tools in software development, as they are the tools that allow the execution of written code, whether through creating a set of machine-readable instructions that can be executed for the given program or by interpreting the source code and executing it. These tools also implement optimizations and quality-of-life features to improve the execution of programs, such as execution speed and ensure that programmers' development is easier and faster. However, there needs to be more information on how these quality-of-life improvements and optimizations impact the energy consumption of the programs.

Some studies try to document the said impacts by examining varied programming languages to understand the factors influencing energy consumption, as evidenced by the studies "Ranking Programming Languages by Energy Efficiency"[39], "Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs"[32], and "Energy Efficiency Across Programming Languages."[38].The main objective of these studies was to create a programming language ranking by energy consumption and to establish a connection between energy consumption and a set of different factors, such as execution time, total memory usage, and peak memory usage.

Nevertheless, as extensive and detailed as the studies mentioned above were, one area is included in future work: the evolution of energy consumption across every iteration of the compilers/interpreters. As mentioned by colleagues, comparing programming languages is exceptionally complex[38]. This complexity is due to various factors, such as implementing the compiler/interpreter, CPU temperatures, total memory usage, and the programs used to test themselves.

With this in mind, this dissertation will describe a rigorous methodology to evaluate a set of 4 Programming Languages, C, C++, Java, and Python, considering the previously mentioned factors. It will also provide insight and in-depth analysis of the state of research on this topic and related topics. For this purpose, we will be able to gather data from a collection of 9 programs from CLBG(The Computer Language Benchmark Game) [18] and answer the following questions:

- RQ1: How can we compare the different versions of the same compiler?
- RQ2: How has the energy consumption, execution speed, and peak memory usage evolved for the compilers?
- RQ3: How do all these metrics relate to one another?
- RQ4: How can we rank the versions for each programming language according to the gathered data? Moreover, if so, what are the results?

These questions will allow us to evaluate the current evolution of energy consumption for these selected programming languages and, hopefully, inspire the people that maintain the current compilers/interpreters to analyze and implement changes in case the languages show upward energy consumption.

With this said, anyone interested in reducing their energy footprint, selecting a language to use in a device with a limited battery, or even in Data Centers can use this dissertation to make an informed decision on what language they should use and the version of said language.

1.2 Dissertation Structure

In the current version of this Dissertation:

- The second chapter is an overview of the current state of research on this topic, as well as highlighting some of the studies that are used as inspiration.
- The third chapter describes the planned methodology to be used for testing the hypothesis presented in this dissertation.
- Finally, the last chapter details the expected results to obtain after testing as well as the potential impact this dissertation could have.

Chapter 2

State of the Art

Research on ranking programming languages according to energy consumption is quite limited, and some aspects still need to be discovered. One such area is how energy consumption varies between compiler/interpreter versions, which will be this dissertation's primary focus.

When researching, I found a few studies on this topic. However, in this section, I will only talk about the ones I found to be the most interesting, thorough, and relevant to my dissertation. These studies are:

- "Energy Efficiency Across Programming Languages." [38]
- "Ranking Programming Languages by Energy Efficiency" [39]
- "Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs" [32]

These are the most current and thorough studies I could find when researching this topic and are the theoretical and practical foundations for developing this dissertation.

The first study's primary focus is to bring attention to energy consumption across programming languages by analyzing the relationship between execution time, memory usage, and energy consumption. To measure information about run time, memory usage, and energy consumption, they used a software-based approach using existing and proven tools to make these measures: RAPL (Running Average Power Limit)[6] and Time (Unix Memory Usage and Execution Time Measure Tool)[17]. The analysis of these measurements employs the Pareto Optimization Algorithm[24] for the relation between time, energy, and memory and the Shapiro-Wilk Test[41] for the normality between peak memory usage and DRAM energy consumption. Some of the findings in this study were that a fast language might not be energy efficient, there is no relation between DRAM energy consumption and memory usage, and no programming language could be the best in all aspects.

The second study is a continuation of the first study by the same authors, with some differences in the types of programs used to test the various factors, the methods to group and analyze the collected data, and the analysis of an extra element which was the total memory usage. Firstly, like the

first study, the programs used to test the energy consumption were from CLBG (Computer Language Benchmark Game). Still, in this second study, there was a focus on seeing the performance of programs with non-optimal performance in mind to mimic more realistic usage conditions and to give a perspective other than the CLBG programs. To achieve this, the researchers also used programs from an open-source educational repository for programming problems called Rosetta Code. Secondly, a clustering method was used to represent how close every language was in terms of energy consumption, execution speed, peak memory usage, and total memory usage. Additionally, the researchers used the Shapiro[41] and Anderson-Darling[37] methods to verify if the data was parametric and to apply the Spearman method[34]. Lastly, this study reached the same conclusions as the previous and also found that the performance of the results obtained from the Rosetta Code was comparable to the ones of the CLBG.

Finally, in the third study, although written by other researchers than the first two, the motivations for their research were very similar. Still, their approaches to the topic were very different. Instead of taking the software approach to measuring energy consumption, in this study, the researchers use a hardware-based approach to measuring energy consumption due to equipment availability and also due to the accuracy of the equipment. Researchers selected the test programs from CLBG because the programs follow the same algorithm across all programming languages implementations of the problem. For some of the compiled languages, they had to run the programs twice to verify the energy impact of the program with and without compilation flags, meaning that in the latter, they did not use any optimization flags unless necessary. Finally, they ended the study by concluding similar things to the other studies, such as that a faster language doesn't imply that energy efficiency will be better. However, they also state that looking only for the most rapid solutions might give an incomplete view of the energy consumption of a language.

As mentioned, these studies have significantly impacted how we perceive energy consumption in programming languages, each with contributions due to their thorough methodology and research. These works serve as an inspiration and cautionary to this topic's intricacies. I want to reiterate that by no means are these the only studies on the subject. Still, these studies distinguish themselves for their thoroughness and for addressing most issues related to the need for knowledge of energy consumption in programming languages.

Other relevant works used for research during the development of this study are the following:

- Analyzing Programming Languages' Energy Consumption: An Empirical Study [25]: The objective of this study was to see how energy consumption differed for small and independent tasks implemented in multiple programming languages.

- E-Debitum: Managing Software Energy Debt [35]: The objective of this study was to energetically catalog code smells for Android and present and validate an open-source tool to calculate the energy debt of apps

- Android App Energy Efficiency: The Impact of Language Runtime Compiler and Implementation [22]: The main goals of this study were to see how energy consumption and

runtime compare if C/C++ was used to develop apps instead of Java. In addition, they verify if using the smartphones multicores could save energy consumption and how recursion and iteration, with and without optimization flags, compare in terms of energy consumption.

- Program Energy Efficiency: The impact of language compiler and implementation choices [20]: With this study, they try to see what sort of impact data structures, libraries, and optimization flags of multiple programming languages have on the energy efficiency of programs.

- Performance exploration of various C/C++ compilers for AMD EPYC processors in numerical modeling of solidification [26]: The goal of this study is quite self-explanatory as they try to evaluate the state of the art for C/C++ Compilers available for the AMD EPYC Rome and Milan and compare the various optimizations made by the multiple compilers subject of study.

- A Comparative Study of Programming Languages in Rosetta Code [36]: This last study tries to evaluate which programming languages are the most concise, the most memory, runtime, and space efficient, and to find which languages are the most prone to failure.

Chapter 3

Methodology

To address the topic of comparing compiler versions by energy efficiency, we must first establish the methods and rules for this experiment. With this in mind, this section of the Dissertation discusses the selected Programming Languages, Compiler Versions, Test Programs, Tools and Collected Data, the Test Framework, Data Processing, and Work Plan. These points will allow us to establish a comprehensive framework for testing the programming languages and their respective compiler/interpreter versions and answer the first investigation question, **RQ1: How can we compare the different versions of the same compiler?**.

3.1 Programming Languages

First, when selecting the number of languages to study, there was a need to limit the number of versions of compilers/interpreters in the study due to the test machine storage space, as some of these compilers can occupy a few Gigabytes of space, and the difficulty of managing the dependencies and issues of the multiple languages. Hence, the number of languages selected for this study is four.

Secondly, at least one language selected needs to use an interpreter to execute code, another uses a virtual machine to compile and execute code, and another that compiles the code and creates a binary executable.

Finally, the language object of study must be popular and currently used. To this effect, we use a popularity ranking website for programming languages called TIOBE [9] to select the study languages: C, C++, Python, and Java.

Four languages were selected instead of three because C and C++ could use the same compilers for compiling programs. Also, the compiler selected for analyzing C/C++ is Clang instead of GCC due to compilation issues regarding older versions of GCC, and overall, the number of older working versions of Clang was greater than GCC versions.

3.2 Compiler Versions

As for the versions selected to test the programs, we started by looking at all releases from 2011 for each language, and from those, we selected the most recent version of each major release since then. The versions that are studied are the following:

Table 3.1: Compiler Versions

Java Versions	Clang Versions	Python Versions
Jdk-6u45	Clang 3.4.2	Python-2.6.9
Jdk-7u80	Clang 3.5.2	Python-2.7.18
Jdk-8u202	Clang 3.6.2	Python-3.0.1
Jdk-8u351	Clang 3.7.1	Python-3.4.10
Jdk-9.0.4	Clang 3.8.1	Python-3.5.10
Jdk-10.0.2	Clang 4.0.0	Python-3.6.15
Jdk-11.0.17	Clang 5.0.2	Python-3.7.16
Jdk-12.0.2	Clang 6.0.1	Python-3.8.16
Jdk 13.0.2	Clang 7.0.1	Python-3.9.16
Jdk-14.0.2	Clang 8.0.0	Python-3.10.10
Jdk-15.0.2	Clang 10.0.0	Python-3.11.2
Jdk-16.0.2	Clang 11.1.0	
Jdk-17.0.6	Clang 12.0.1	
Jdk-18.0.2.1	Clang 13.0.1	
Jdk-19.0.2	Clang 14.0.0	
Jdk-20.0.1	Clang 15.0.6	

It is also worth mentioning that some compiler/interpreter versions have been updated between the time of installation and the time of writing. However, the realization of the study did not consider these newer versions.

Finally, we use the same flags used in CLBG to compile and execute the benchmark programs, specifically languages like C/C++. We use these optimization flags to make sure that the programs are being executed in the same way as in CLBG but also to guarantee that all versions of the same compilers are on the same playing field. Also, since this project aims to compare the performance between versions of the same compiler and not between languages, we use these optimization flags even if it would give them an edge over the other languages. The impact of these flags could be a subject for a continuation of this study.

3.3 Test Programs

To test the energy efficiency of the compilers/interpreters, we need a set of programs equivalent to all programming languages studied in this dissertation. To this effect, we use a set of programs with the same algorithm and complexity obtained from CLBG (Computer Language Benchmark Game) [18] for testing purposes.

The CLBG Initiative mentioned above has developed a framework for testing and comparison of multiple programming languages using a collection of general programming problems. The framework developed by the CLBG Initiative measures various metrics during the execution of the abovementioned problems, allowing for a comparison of execution speed and memory consumption, for example.

Due to the unavailability of some programs, such as the Simple and Too Simple problems for C++, we use only ten problems for testing out of all the programs available on CLBG. These programs are in table 3.2, along with a short description of the problem and input size used for testing.

Table 3.2: Test Programs

Name	**Description**	**Data Size**
binary-trees	Allocate and deallocate many many binary trees	21
fannkuch-redux	Indexed-access to tiny integer-sequence	12
fasta	Generate and write random DNA sequences	25.000.000
k-nucleotide	Hashtable update and k-nucleotide strings	25.000.000
mandelbrot	Generates a Mandelbrot set	16.000
n-body	Double-precision N-body simulation	50.000.000
pi-digits	Calculates all digits in pi till the nth position	10.000
regex-redux	Match DNA 8-mers and substitute magic patterns	5.000.000
reverse-complement	Converts a DNA sequence into it’s reverse-complement	25.000.000
spectral-norm	Eigenvalue using the power method	5.500

As expected, not all programs can be compiled or executed by all versions of the compilers/interpreters. So before any tests, we made an effort to maximize the number of compiler/interpreter versions that could compile and execute the selected programs, by changing the versions of the benchmark programs. In order to do this, we created a set of rules to select the versions of the programs, rules which are the following:

- If all the versions of the compiler/interpreter can compile and execute it, then there is no need no maximize it.
- If an older program version has more compiler/interpreter versions that can compile and execute it, it becomes the test program for all versions.
- In case all older program versions have the same number of working compiler/interpreter versions currently selected version, the fastest version, according to CLBG, becomes the test program.

Following this set of rules, the versions of the selected program, as numbered in CLBG, are the ones present in table 3.3.

Even though all the programs used for benchmarking are functionally identical, they should not be used to compare programming languages as there are more efficient versions of some benchmarks, which could induce false statements about which programming languages are better when

Table 3.3: Program Versions

Programs	**Java**	**C**	**C++**	**Python**
binary-trees	7	5	8	5
fannkuch-redux	3	5	7	4
fasta	6	9	6	3
k-nucleotide	6	1	8	8
mandelbrot	6	9	9	7
n-body	5	9	7	2
pi-digits	1	2	5	4
regex-redux	6	5	6	1
reverse-complement	6	8	6	4
spectral-norm	3	8	1	8

the objective is to compare the different versions between compilers of said programming languages.

After some preliminary testing and using the program versions that maximized the number of working compiler/interpreter versions, the resulting compilation and execution tables are shown if Fig. 3.1 for Java, Fig.3.2 for C, Fig.3.3 for C++ and Fig.3.4 for Python.

In the previously mentioned tables, we have two columns for each benchmark. The first column details if the corresponding version can compile or not the benchmark using either T for True and F for False, while the second details if the version can execute it or not using the same system as the first column. For added visibility, if a cell has F in it, it is then colored with red, and if the cell has T in it, it is colored with green.

Versions	Binary Trees		Fannkuch Redux		Fasta		K-Nucleotide		Mandelbrot		N-Body		Pi-Digits		Regex-Redux		Rev-Comp		Spectral Norm	
	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run
6	F	F	T	T	F	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F
7	F	F	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	F	F
8.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	F	F
8.1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	F	F
9	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	F	F
10	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
11	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
12	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
13	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
14	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
15	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
16	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
17	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
18	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
19	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
20	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Figure 3.1: Java Compilation and Execution

Versions	Binary Trees		Fannkuch Redux		Fasta		K-Nucleotide		Mandelbrot		N-Body		Pi-Digits		Regex-Redux		Rev-Comp		Spectral Norm	
	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run
3.4	T	T	T	T	T	T	T	T	T	T	F	F	T	T	T	T	T	T	F	F
3.5	T	T	T	T	T	T	T	T	T	T	F	F	T	T	T	T	T	T	F	F
3.6	T	T	T	T	T	T	T	T	T	T	F	F	T	T	T	T	T	T	F	F
3.7	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.8	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
4.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
5.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
6.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
7.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
8.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
9.0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
10.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
11.1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
12.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
13.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
14.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
15.0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Figure 3.2: C Compilation and Execution

Versions	Binary Trees		Fannkuch Redux		Fasta		K-Nucleotide		Mandelbrot		N-Body		Pi-Digits		Regex-Redux		Rev-Comp		Spectral Norm	
	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run
3.4	F	F	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
3.5	F	F	T	T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	T	T
3.6	F	F	T	T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	T	T
3.7	F	F	T	T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	T	T
3.8	F	F	T	T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	T	T
4.0	F	F	T	T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	T	T
5.0	F	F	T	T	T	T	F	F	T	T	F	F	T	T	T	T	T	T	T	T
6.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
7.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
8.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
9.0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
10.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
11.1	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
12.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
13.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
14.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T
15.0	T	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T	T

Figure 3.3: C++ Compilation and Execution

Versions	Binary Trees		Fannkuch Redux		Fasta		K-Nucleotide		Mandelbrot		N-Body		Pi-Digits		Regex-Redux		Rev-Comp		Spectral Norm	
	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run	Comp	Run
2.6	T	F	T	F	T	F	T	F	T	F	T	T	T	F	T	T	T	F	T	F
2.7	T	F	T	F	T	F	T	F	T	F	T	T	T	F	T	T	T	F	T	F
3.0	T	F	T	F	T	T	T	T	T	F	T	T	T	T	T	T	T	F	T	F
3.4	T	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.5	T	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.7	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.8	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.9	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.10	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
3.11	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Figure 3.4: Python Compilation and Execution

Most of the versions of the compilers can indeed compile and generate a working executable of the selected programs and translate the source code successfully without issues in the case of interpreters. However, for the C++ program of K-nucleotide, compilation and execution were not achieved due to an issue finding the file "assoc_container.hpp" in libstdc++. Even after adding the flags to include it in the compilation, it would still be unsuccessful due to the said missing file. Initially, there was also an attempt to study the Clang v9.0.1 version alongside the others. However, every compilation attempt of a program would cause a segmentation fault, and thus it was removed from the study altogether.

The other programs that failed to compile/run were due to missing functionalities of that program version or unsupported syntax in the program's code.

3.4 Tools and Collected Data

This section discusses why data related to execution time, peak memory usage, and energy consumption is collected, the tools used, and the reason for using the selected tools to reach any conclusion related to the evolution of the mentioned languages and versions.

First, there are two ways of collecting data related to energy consumption: measuring the power consumption from the hardware directly or using software to estimate the energy consumption. Due to the lack of equipment for measuring hardware energy consumption, there was a decision made to use the software tool provided by Intel: RAPL(Running Average Power Limit), which has been tested and proven to be an accurate measuring tool for energy consumption [31].

For measuring the execution time of the test programs, there was a choice between measuring the CPU time for the task or the wall-clock time (aka.: real elapsed time). Since CPU time depends on the architecture, the run-time factors, and is actively measured only when the CPU is working, it made sense not to use this time measurement because the CPU time might be higher or lower

than the program's elapsed time. Hence time measurements will be done using the time function in Unix systems, as they provide a reliable to measure the elapsed time for execution and can also be used to collect data related to peak memory usage.

3.5 Test Framework

To automate the testing of the multiple versions of the compilers, we developed a framework in Python. In this section of the dissertation, we will discuss some of the development decisions and the overall structure of the framework.

The framework's structure is relatively simple, containing a few Python scripts that do most of the heavy lifting of the testing. However, there are a few things that need addressing relatively to the folder structure, makefiles, execution of the programs, CPU Temperature and data collection and saving

For the folder structure of this project, each language has its folder, and inside this folder, they contain sub-folders related to each of the test programs. To allow simplicity and modularity, each of these subfolders contain a makefile that has the following methods:

- Compile: creates subfolders for the compiler/interpreter versions specified in the makefile, compiles the test program with each version, and copies the executable/binary into the correct folder. As for Python, it only creates the subfolders and copies the program into said folder.
- Run: Runs the executable without any measures
- Measure: This runs the executable through "main.py" and measures execution speed, package energy consumption, dram energy consumption, and peak memory usage.

This file structure and makefile templates allow future users to add more languages for further testing and more test programs. Nevertheless, this framework can still be improved, especially when creating new makefiles, which are laborious and need more automation.

In order to measure the performance of the programs, the framework has implemented two methods that allow the collection of data, these methods being "collect_single" and "collect_multiple," which refer to whether data collection is at the start and end of the execution of the program or along the execution of the program with the help of an extra thread.

```
1 def collect_multiple(self):
2         ...
3       self.meter.begin()
4       exec = Thread(target=self.execute_command)
5       exec.start()
6
7       while exec.is_alive():
8         #For small duration programs freq should be very low or it wont measure
              many time
```

```
9             time.sleep(self.freq)
10            self.meter.end()
11            ...
12
13            ...
14
15    def collect_single(self):
16
17        ...
18
19             self.meter.begin()
20
21             process = self.execute_command()
22
23             self.meter.end()
24
25        ...
```

For this experiment, we use the single-threaded collection as some preliminary testing of the multi-threaded showed that the single-threaded method had better performance in all metrics. In some preliminary tests where run execution was lower than a second, we could see a significant increase in execution time and energy consumption,as present in table 3.4, thus being excluded.

Table 3.4: C N-Body Single-Thread vs Multi-Thread

-	Duration(μs)	Package (μJ)	Dram (μJ)	Peak RSS (kB)
Single Threaded	2121,7042	30285,4	1135,4	1717,2
Multi Threaded	10502,2386	76946,8	7446,2	1717,6
Comparison	394,99%	154,07%	555,82%	0,02%

To even further reduce the energy consumption of the entire system, only the essential services of the OS were kept running, these being the ones on table 3.5.

Aside from keeping these services alive during the execution, we also turn off any Wifi and Bluetooth connections the test machine might have.

As for the CPU temperature, because tests are automated, there was the need to implement a system that would allow the CPU to heat up and cool down its temperature in between tests, as CPU temperature impacts the package energy consumption made by RAPL [31]. In order to heat the CPU, the framework spawns as many threads as there are cores and proceeds to execute a round-robin on each of these threads until the CPU reaches its minimum threshold temperature. Contrarily, to cool down the CPU, the framework sleeps for a set amount of time until the CPU temperature goes below the maximum temperature threshold. In these experiments, the temperature threshold was between 40ºC and 45ºC, and the interval used for the sleep was 0.01 seconds. Also, we keep the previous temperature measure to ensure that the temperature reached was not just a temperature spike.

Table 3.5: Services kept alive and reasoning

Service	Reasoning
acpid	"is designed to notify user-space programs of ACPI events." [1]
apparmor	"is a Linux Security Module implementation of name-based mandatory access controls." [14]
apport	intercepts Program crashes, collects debugging information about the crash and the operating system environment, and sends it to bug trackers in a standardized form. [15]
cron	is a job scheduler on Unix-like operating systems. [10]
dbus	"a library that provides one-to-one communication between any two applications" [11]
gdm	is a display manager (a graphical login manager) for the windowing systems. [12]
irqbalance	"is to distribute hardware interrupts across processors on a multiprocessor system in order to increase performance." [13]
kerneloops	is a program that collects kernel crash information and then submits the extracted signature to the kerneloops.org website for statistical analysis and presentation to the Linux kernel developers. [3]
kmod	"is a multi-call binary which implements the programs used to control Linux Kernel modules." [4]
plymouth-log	"is considered to be the "owner" of the console device (/dev/console) so no application should attempt to modify terminal attributes for this device at boot or shutdown." [16]
procps	report a snapshot of the current processes. [16]
udev	"provides a dynamic device directory containing only the files for actually present devices." [5]

Finally, the DataRAPL class, responsible for saving the data, creates CSV with all pertinent information such as duration, package energy consumption, dram energy consumption, peak memory usage (RSS), and initial and final temperatures.

For more information about the source code of the the developed Framework, see the following github link[8].

3.6 Data Aggregation and Cleaning

With the data collected from the experiments, we first perform three normality tests, the Shapiro-Wilk test[27], the Anderson-Darling test[40], and the D'Agostino-Pearson test[23], for each metric to confirm whether the data we are working with is parametric. For accuracy, we only consider data normally distributed when all three tests pass, which is essential to perform the Pearson Correlation Coefficient test between two normally distributed variables.

With said normality tests performed, we then perform the Spearman Correlation Coefficient and Pearson Correlation Coefficient tests[19] for each set of two metrics. We performed these tests to find the correlation between execution speed, package energy consumption, DRAM energy

consumption, and peak memory usage. However, for the Pearson test, we also need to verify the homoscedasticity between the variables we are trying to establish a correlation, an assumption verified using the Bartlett test[2] for Homogeneity of Variances.

Contrary to the Pearson test, the Spearman can be directly applied as it only assumes a monotonic relationship between variables, and data needs to be continuous or ordinal.

As for creating a ranking of the compiler/interpreter versions for each programing language and each benchmark, we rank the various metrics according to the versions that could execute said benchmarks. Using the ranks obtained for each test program and language, we then proceed to create a global ranking for each language using the following formula:

$$Score = 200 - (\sum_{i=1}^{n} rank(i, v))$$

, where n is the number of benchmarks, and v the version to rank

Using this formula, we can accurately reward and punish according to the performance of each benchmark. However, not all benchmarks and versions are used for the ranking, as not all versions can run specific benchmarks, and as such, if we were to use said benchmarks and versions, we could negatively impact the accuracy of the ranking. With this in mind, we select the versions and benchmark for the ranking by maximizing the product between the number of working versions and the number of benchmarks. For each language, the used benchmarks for the ranking were the following 3.6:

Table 3.6: Benchmarks used for ranking by language

Language	**Benchmarks**
Java	Binary Trees, Fannkuch Redux, Fasta, K-Nucleotide, Mandelbrot, N-Body, Pi-Digits
C	All benchmarks were used
C++	Fannkuch Redux, Fasta, Mandelbrot, Pi-Digits, Rev-Comp, Spectral-Norm
Python	All benchmarks with the exception of binary trees

Chapter 4

Results

4.1 Presentation of Results

In this section, we present the obtained values of the experiments for the Binary Trees, Fannkuch Redux, and Fasta, as well as the general ranking of the versions for each language. We only display the first three benchmark programs due to the project's sheer amount of data.

The full results and data can be accessed through the following Google Sheets Page [7] and is the primary resource used to establish any conclusions. In this sheet, you can see the results for the remaining benchmarks, the rankings for each metric by benchmark and a general ranking, and the results for the Spearman and Pearson Correlation Coefficients.

Also, to reduce the amount of data even further, the presented graphs use the total energy used instead of the energy consumed by the package and the energy consumed by the DRAM separated.

For each benchmark, we present two simple line graphs, the first one displaying the evolution of runtime and the total energy consumed and a separate graph displaying the evolution of the peak memory consumption. Then for each Programming Language, we also display two line graphs, the first displays each version of the compiler/interpreter and their rank in terms of runtime, package energy consumption, DRAM energy consumption, and total energy consumption, while the second one display each version of the compiler/interpreter and their rank in terms of peak memory consumption. This separation was mostly done to improve the visibility of the graphs and also because, when compared to duration and all energy metrics, we could see different behavior.

In some general rankings, such as 4.7, 4.23,and 4.31, we can see that one line is missing. This missing line is not a mistake because the Package Energy Consumption rank overlaps with the Total Energy Consumption Rank.

4.1.1 Java Results

For Java, the individual benchmark results obtained are present in the graphs for Binary Trees(4.1, 4.2), Fannkuch-Redux(4.3, 4.4), and Fasta(4.5, 4.6) and the general rankings are present in these 4.7,4.8.

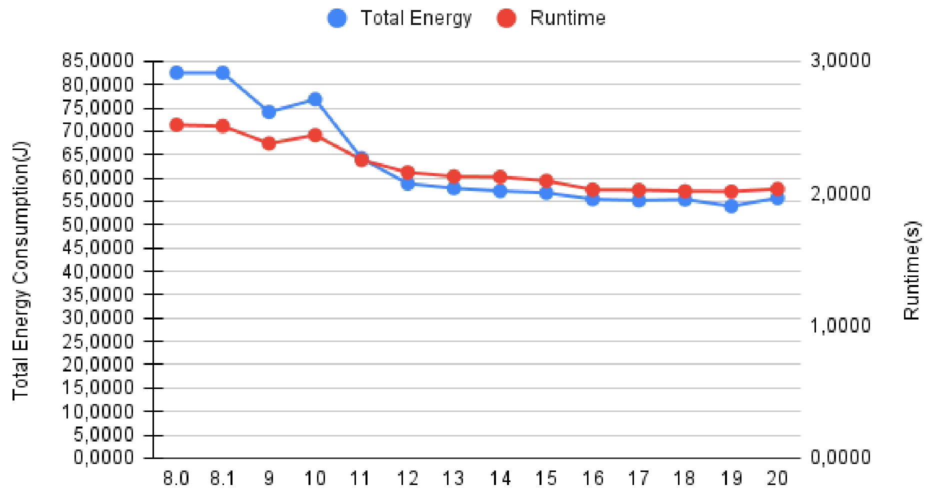

Figure 4.1: Java Results Binary Trees

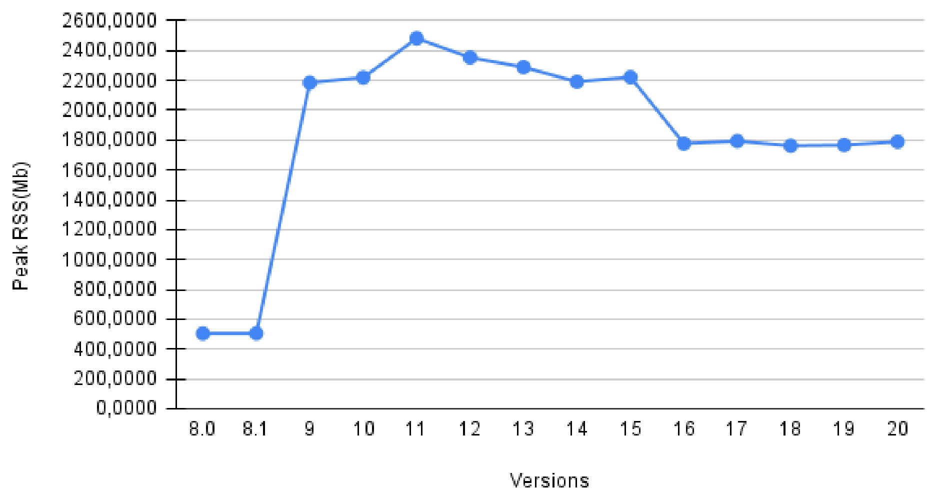

Figure 4.2: Java Peak RSS Performance Binary Trees

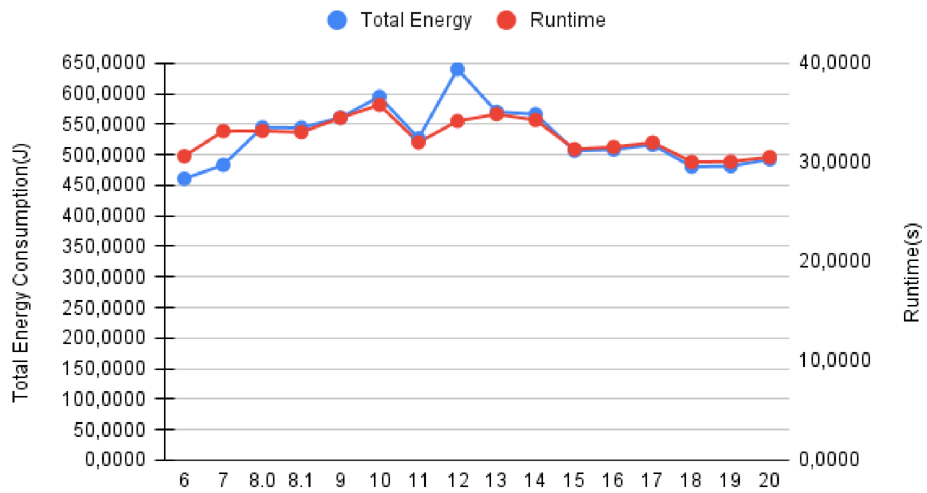

Figure 4.3: Java Results Fannkuch Redux

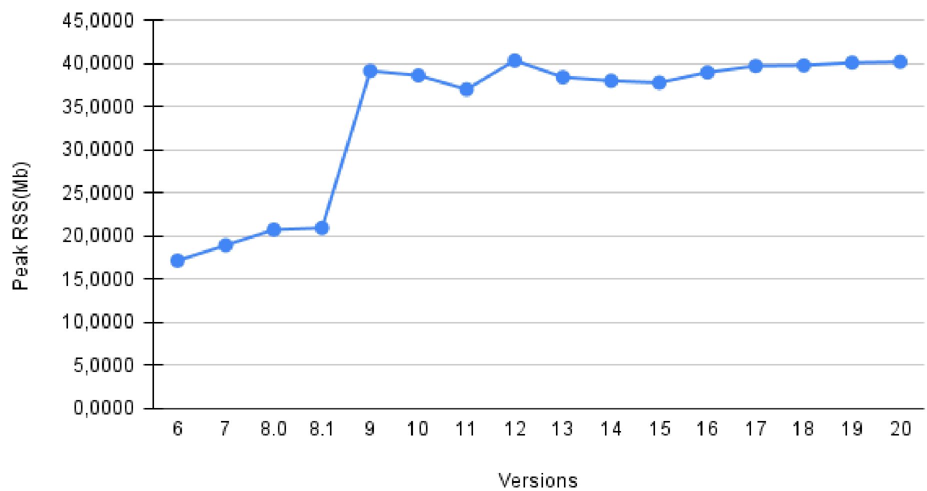

Figure 4.4: Java Peak RSS Performance Fannkuch Redux

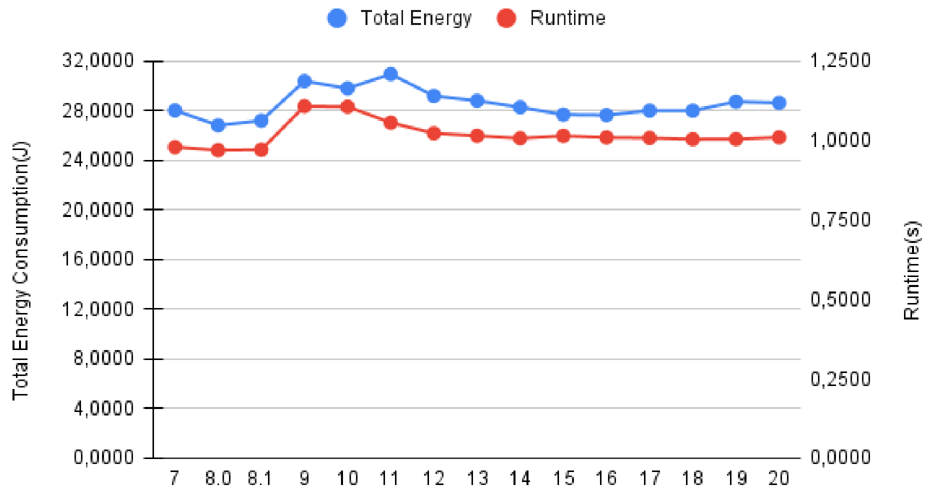

Figure 4.5: Java Results Fasta

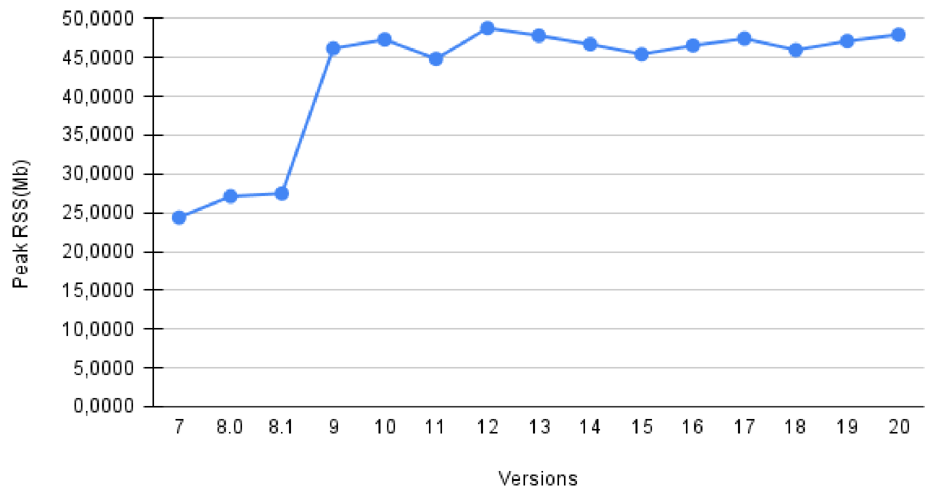

Figure 4.6: Java Peak RSS Performance Fasta

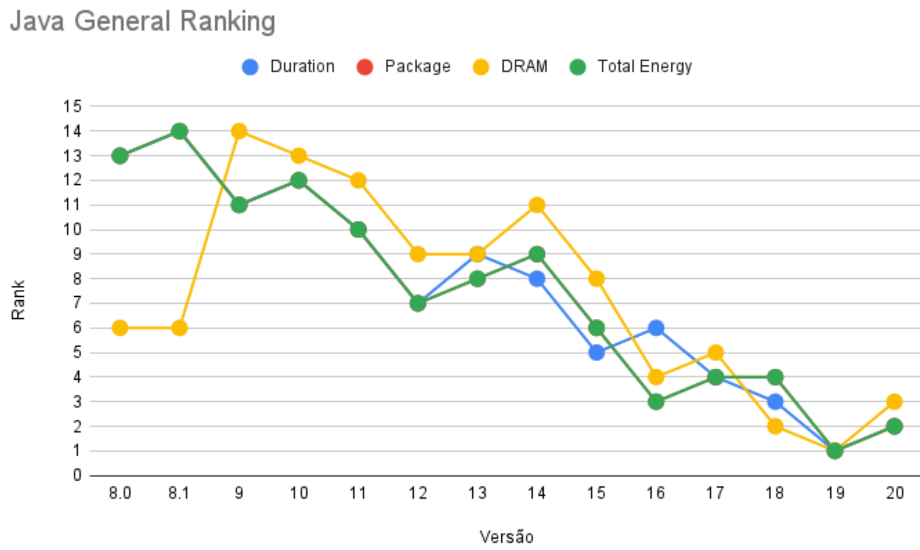

Figure 4.7: Java General Rank

Figure 4.8: Java Peak RSS Rank

4.1.2 C Results

For C, the individual benchmark results obtained are present in the graphs for Binary Trees(4.9, 4.10), Fannkuch-Redux(4.11, 4.12), and Fasta(4.13, 4.14) and the general rankings are present in these 4.15,4.16.

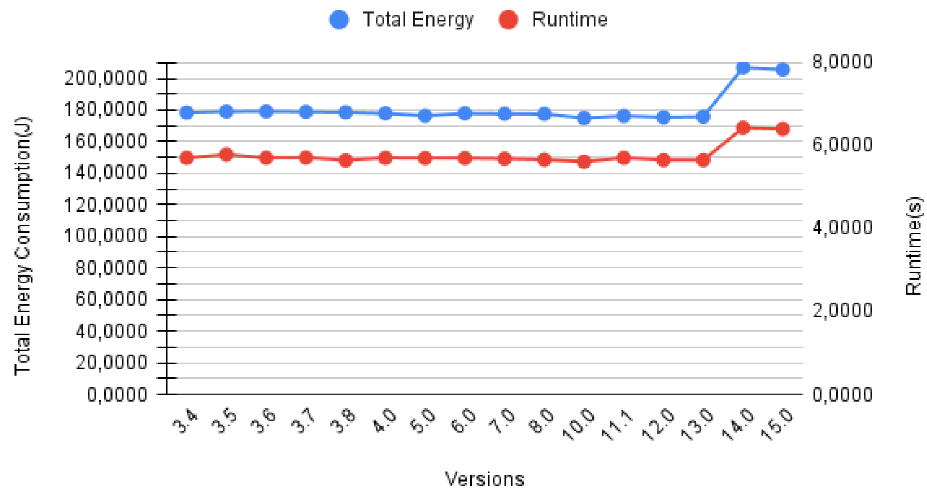

Figure 4.9: C Results Binary Trees

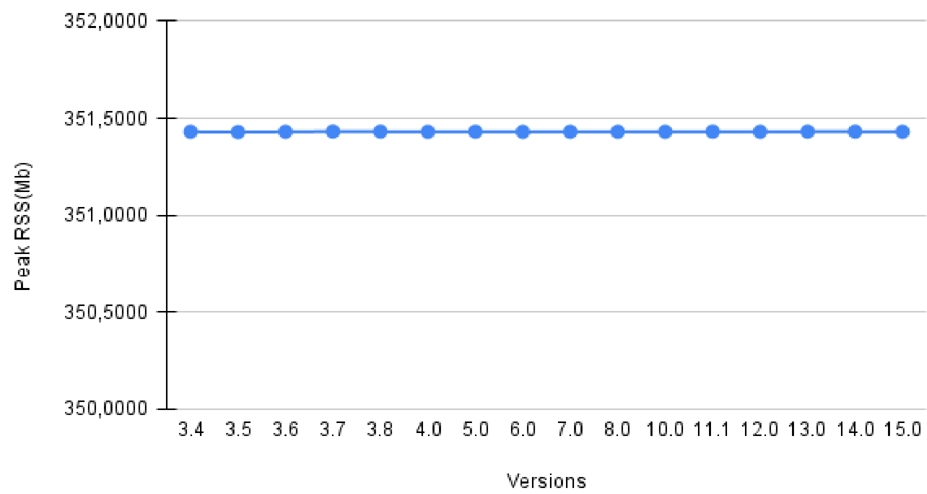

Figure 4.10: C Peak RSS Performance Binary Trees

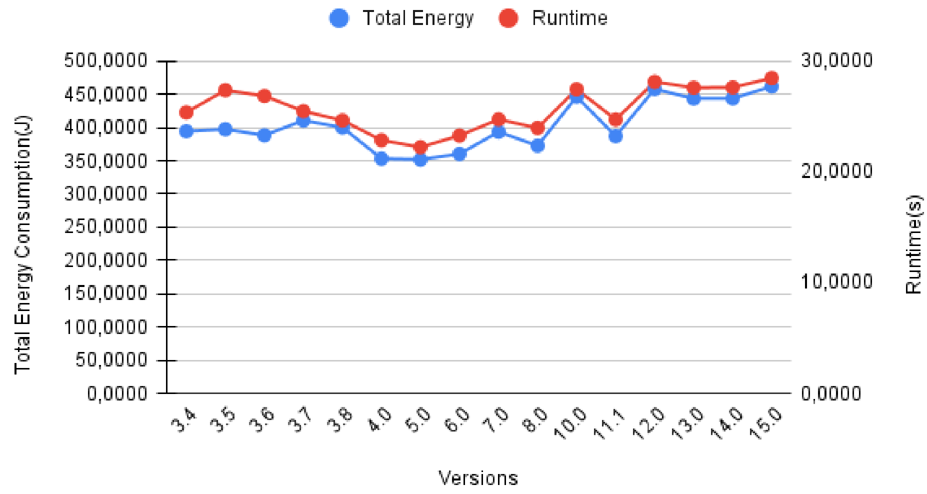

Figure 4.11: C Results Fannkuch Redux

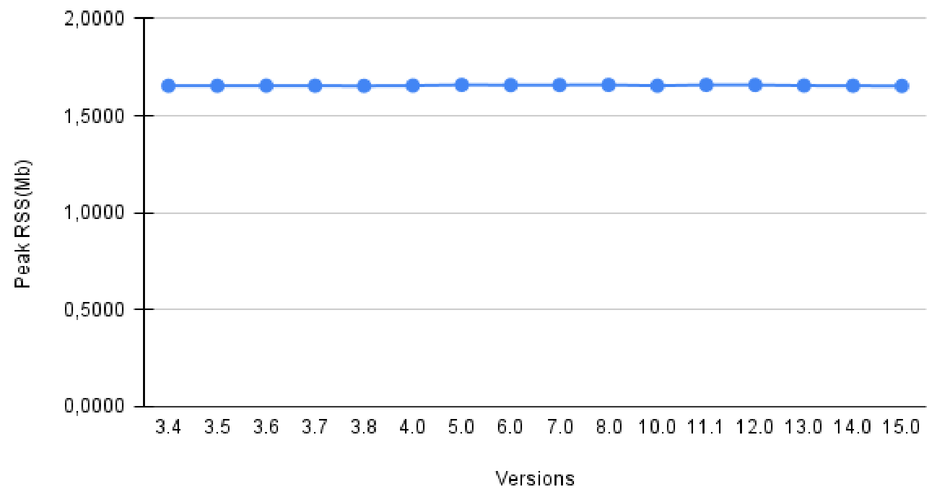

Figure 4.12: C Peak RSS Performance Fannkuch Redux

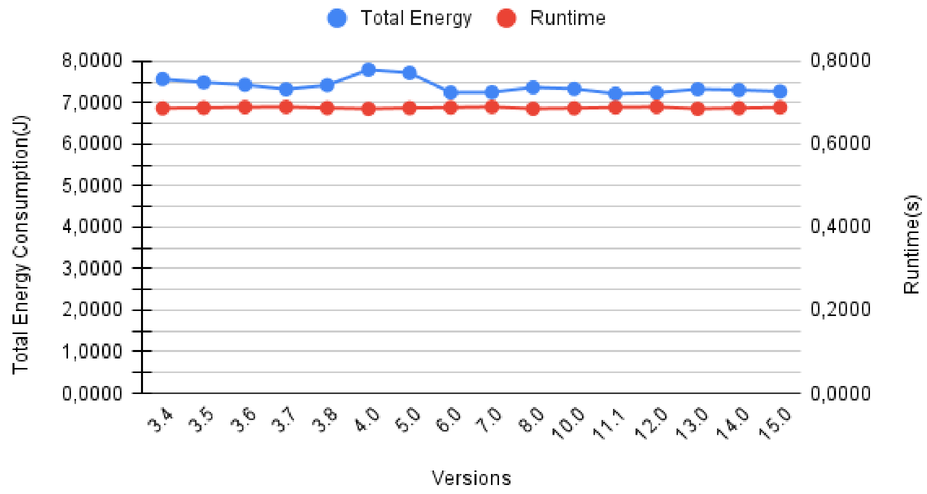

Figure 4.13: C Results Fasta

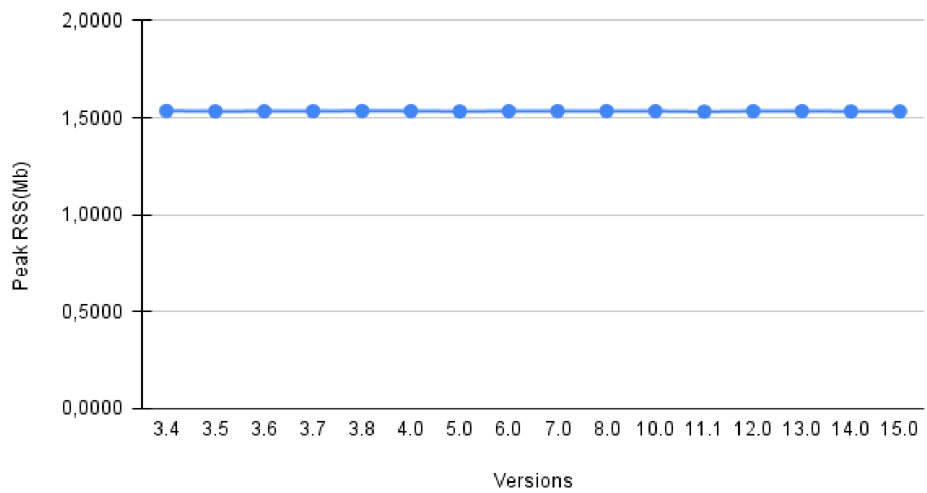

Figure 4.14: C Peak RSS Performance Fasta

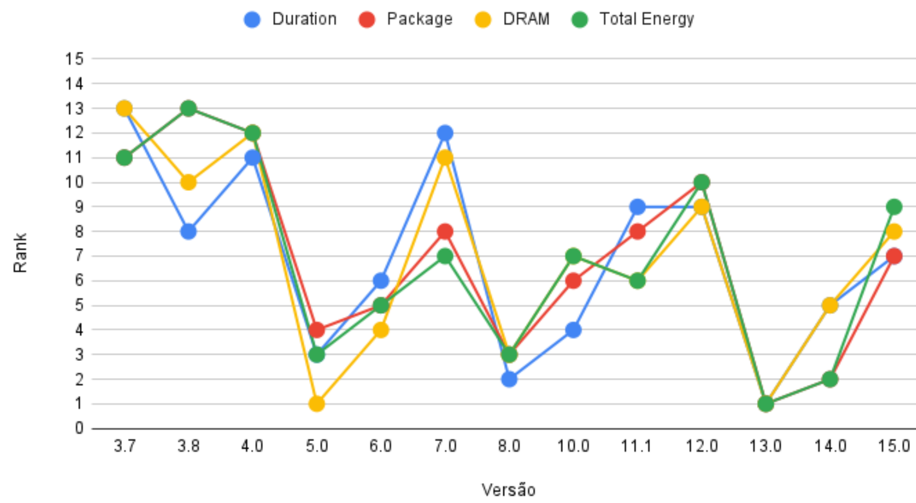

Figure 4.15: C General Rank

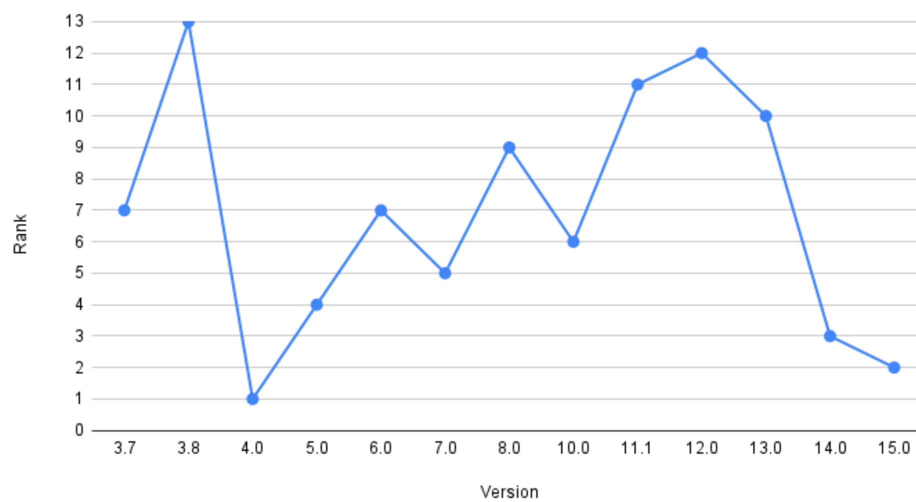

Figure 4.16: C Peak RSS Rank

4.1.3 C++ Results

For C++, the individual benchmark results obtained are present in the graphs for Binary Trees(4.17, 4.18), Fannkuch-Redux(4.19, 4.20), and Fasta(4.21, 4.22) and the general rankings are present in these 4.23,4.24.

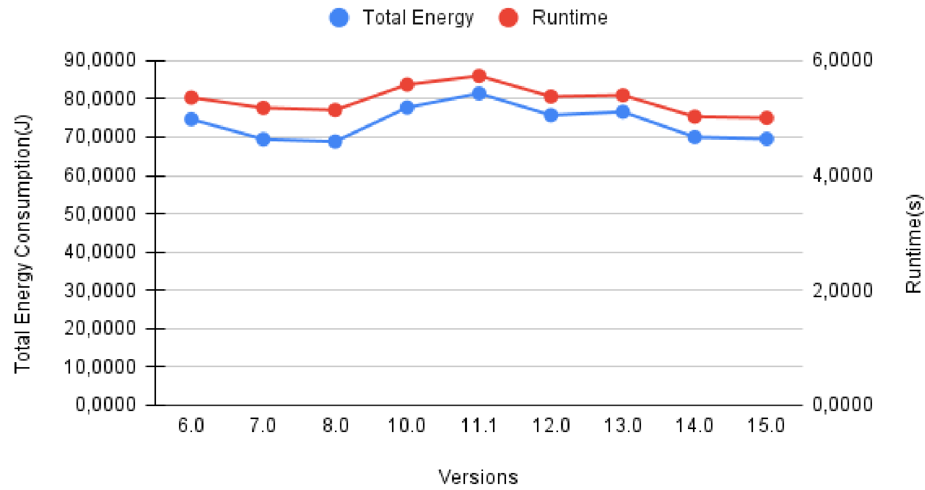

Figure 4.17: C++ Results Binary Trees

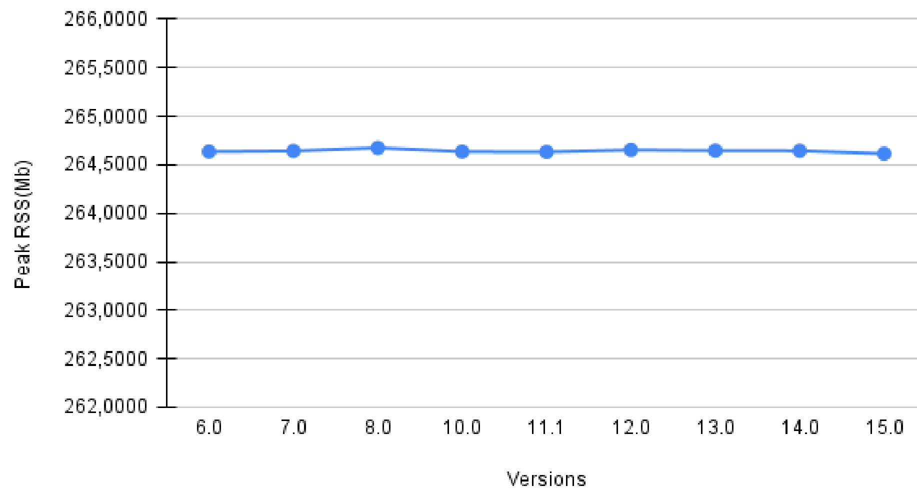

Figure 4.18: C++ Peak RSS Performance Binary Trees

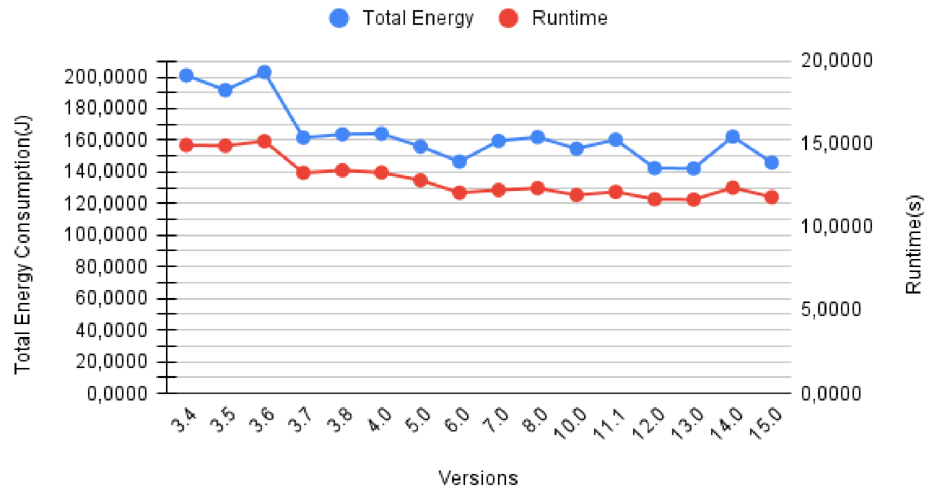

Figure 4.19: C++ Results Fannkuch Redux

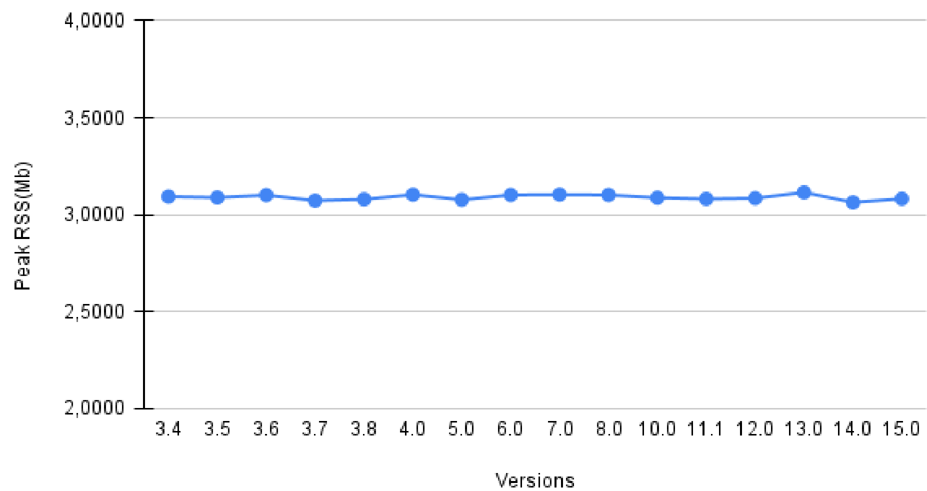

Figure 4.20: C++ Peak RSS Performance Fannkuch Redux

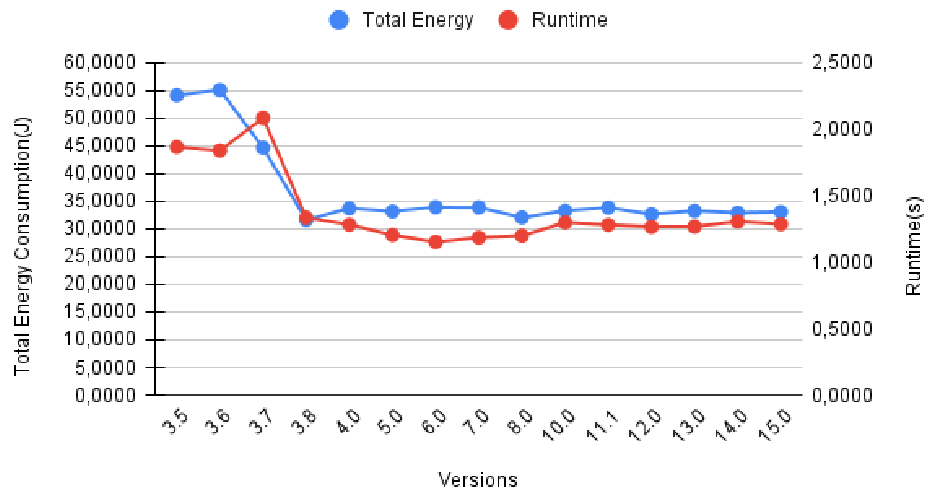

Figure 4.21: C++ Results Fasta

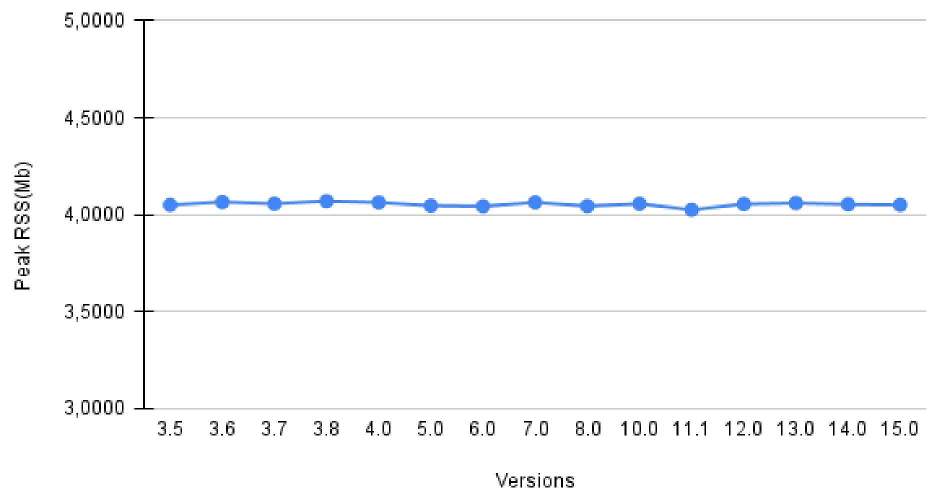

Figure 4.22: C++ Peak RSS Performance Fasta

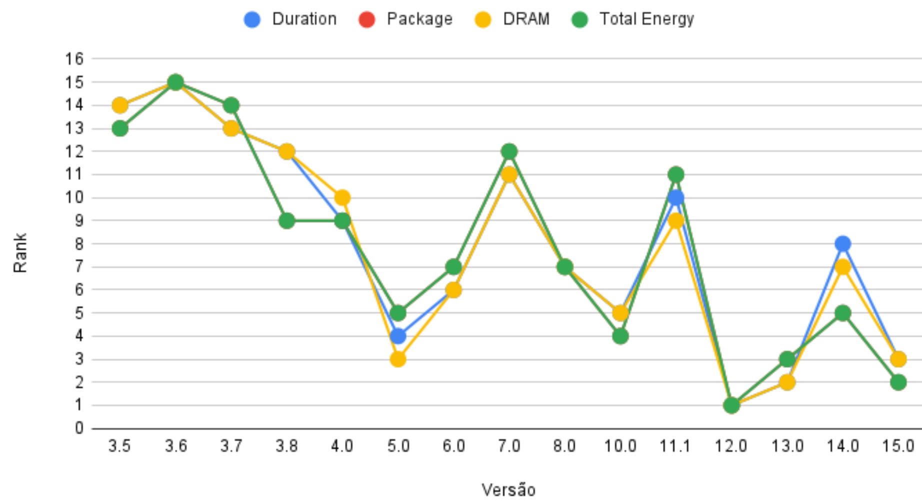

Figure 4.23: C++ General Rank

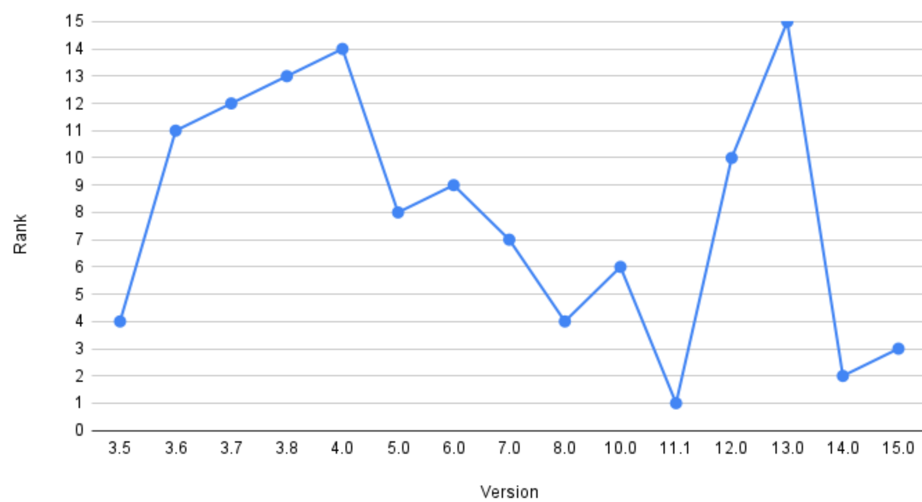

Figure 4.24: C++ Peak RSS Rank

4.1.4 Python Results

For Python, the individual benchmark results obtained are present in the graphs for Binary Trees(4.25, 4.26), Fannkuch-Redux(4.27, 4.28), and Fasta(4.29, 4.30) and the general rankings are present in these 4.31,4.32.

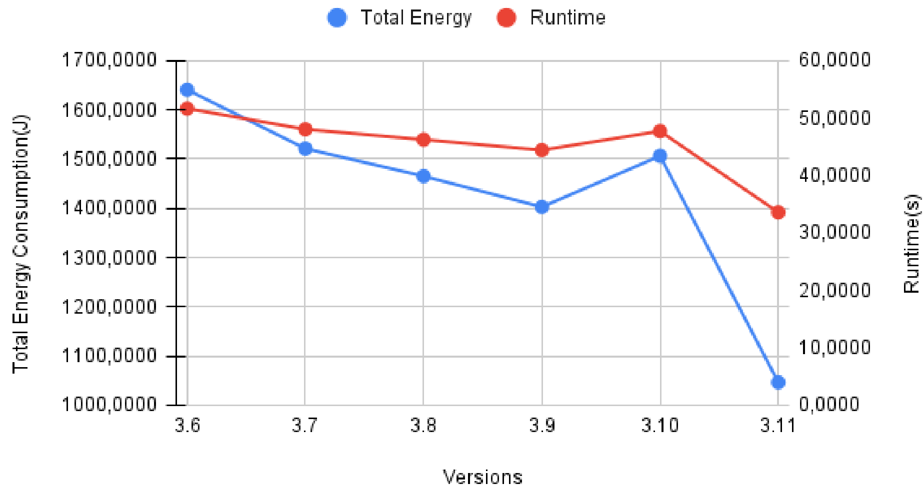

Figure 4.25: Python Results Binary Trees

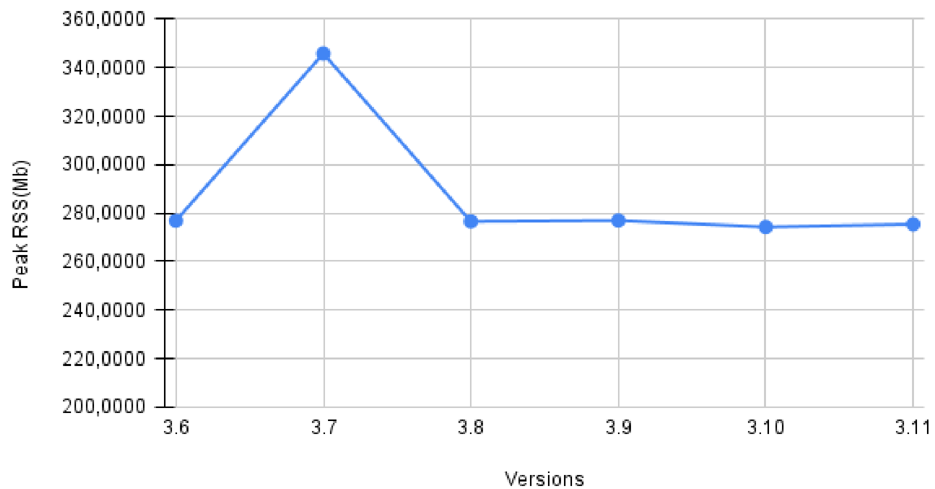

Figure 4.26: Python Peak RSS Performance Binary Trees

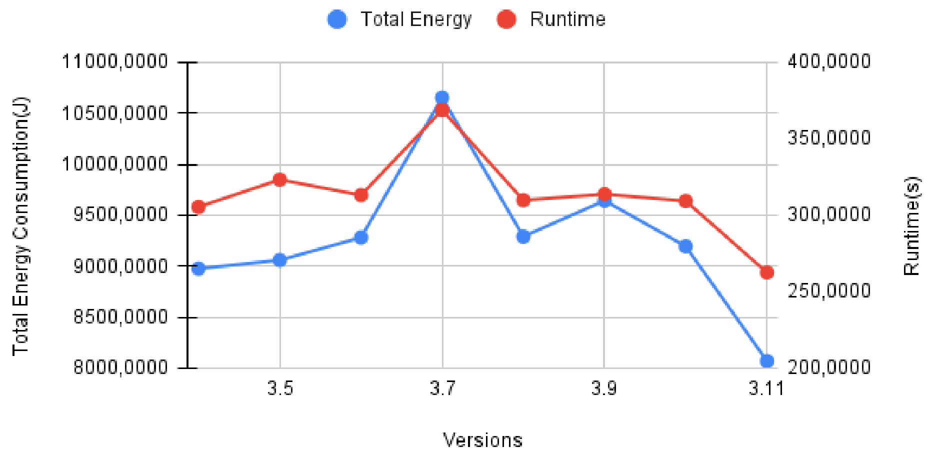

Figure 4.27: Python Results Fannkuch Redux

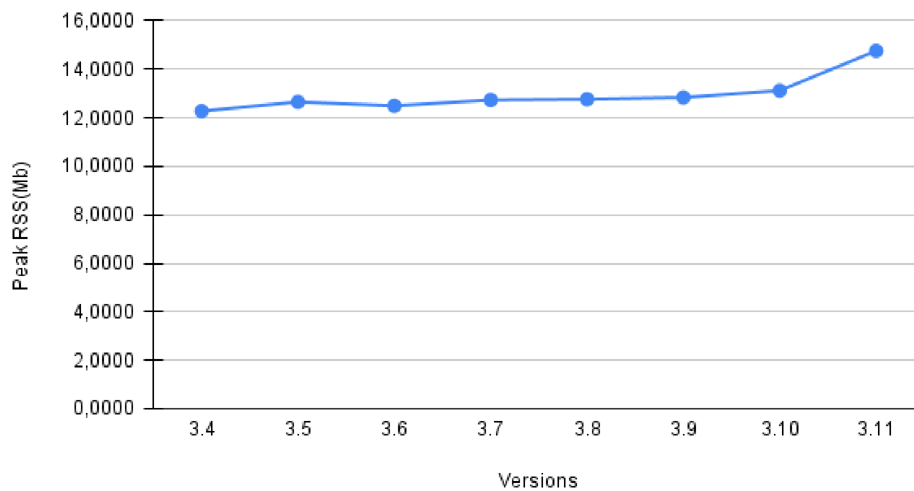

Figure 4.28: Python Peak RSS Performance Fannkuch Redux

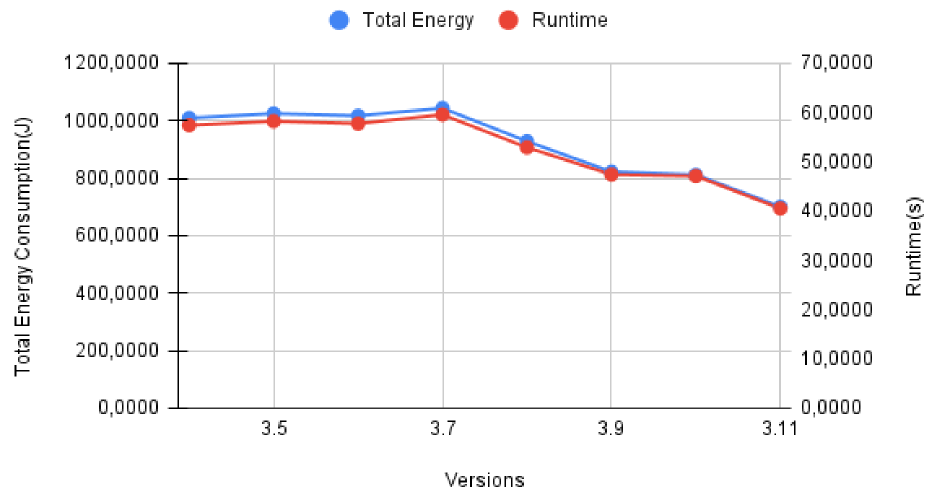

Figure 4.29: Python Results Fasta

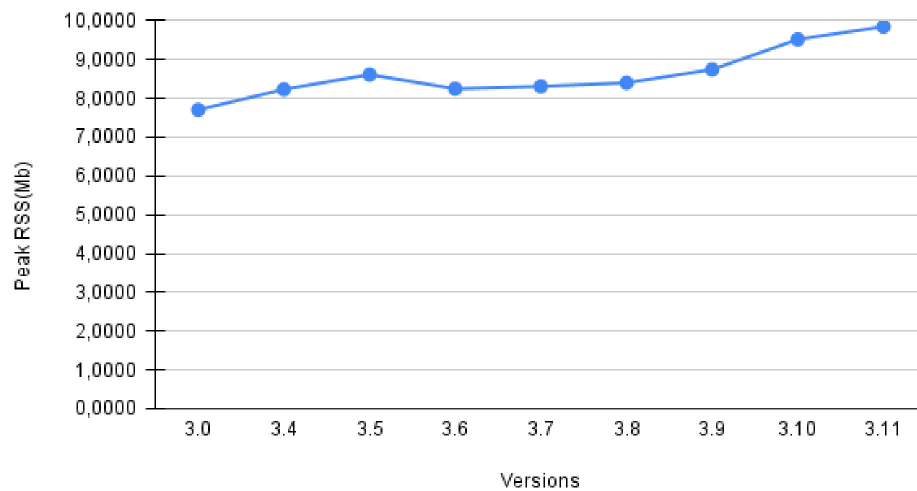

Figure 4.30: Python Peak RSS Performance Fasta

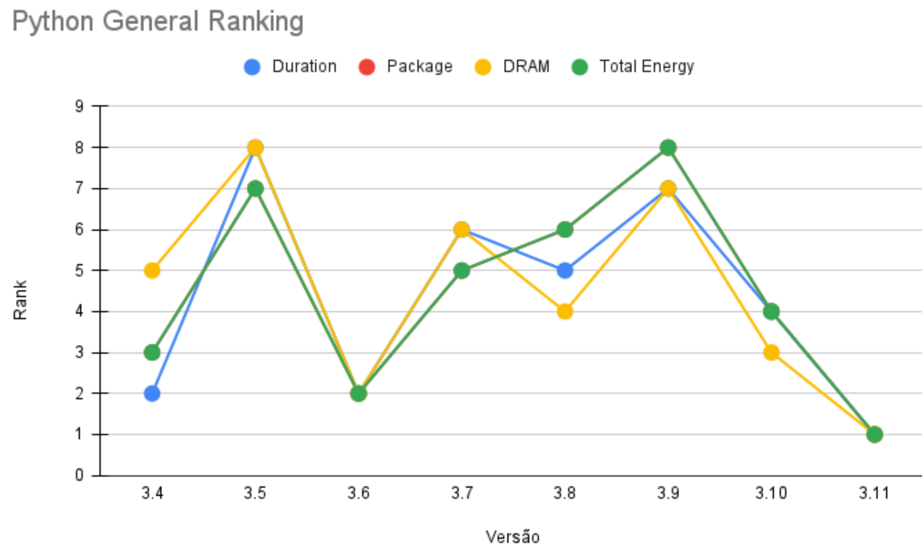

Figure 4.31: Python General Rank

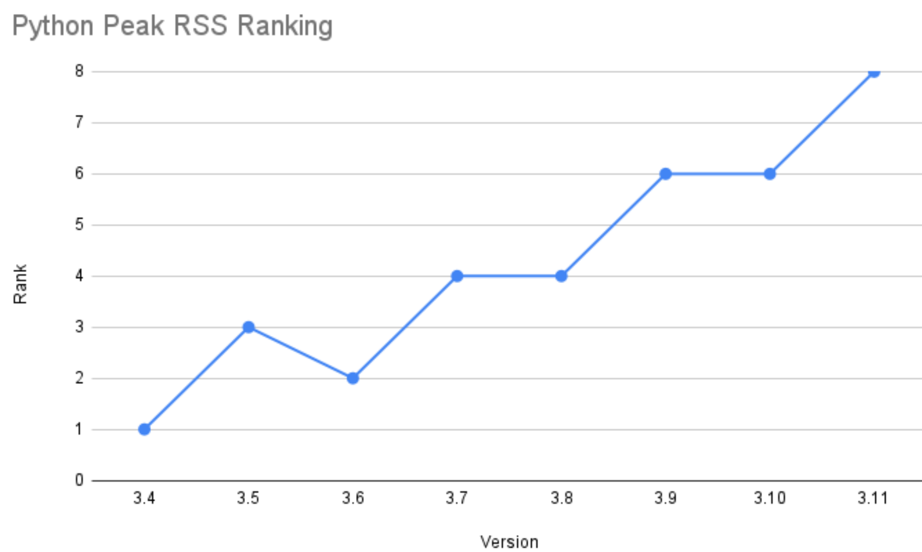

Figure 4.32: Python Peak RSS Rank

Before discussing the results, we would like to reiterate that this project aims to compare the performance between versions of the same compiler and not between programming languages. Some benchmark results showed that Java could outperform both C and C++, binary trees, for example. Even though it was proved that C/C++ has overall better performance than Java [39], results like these are perfectly normal, which can be confirmed by comparing these results with the ones presented on the CLBG website. If we compare the results of both, most relations between programming languages stay the same, even if the metrics are not, meaning if, for one CLBG benchmark, the performance from best to worst goes Java, C, C++, and Python, we can see a mirror of these results in the ones collected. The only exceptions to these were the k-nucleotide and mandelbrot benchmarks. However, this can be explained by the difference in software, hardware, operating system, and environmental conditions used for this project and the ones for the CLBG website as well as the fact that these programs were developed with a specific compiler version in mind. The above arguments can also explain the difference in metrics between the CLBG and the obtained results. In conclusion, the results obtained are more than valid to establish any conclusions, mainly because we are not comparing the performance between the languages but between the versions of the same compiler and because most relations between programming languages mirror the ones from CLBG, even if the collected metrics differ from the ones in CLBG.

4.2 Discussion of Results

In this section of the dissertation, we will discuss the results obtained and try to answer the proposed investigation questions. Starting with:

RQ2 and RQ3: How has the energy consumption, execution speed, and peak memory usage evolved for the compilers? How do all these metrics relate to one another?

From the individual benchmarks, we suspect that energy efficiency is not one of the top priorities for compiler/interpreter developers, as the benchmark performance can vary from version to version, not following a specific trend. However, if we consider the general ranking, we can notice a gradual decrease, with some spikes in some versions, in energy consumption and runtime speed, except for C, which is understandable due to the maturity of the language, which can cause seemingly random ranks in their general ranking.

Overall, C is the language that changes the least according to the individual benchmarks, with little to no change in energy consumption or runtime performance, with some exceptions, such as the results from Fannkuch Redux, which see the values vary a bit through all versions, and Mandelbrot, who we can see a decrease in energy consumption and program runtime along the versions.

Another possible conclusion from the general rankings and the individual benchmarks is that execution time is directly tied to the package energy consumption, which would make sense due to the formula for energy consumption $E = P * \delta t$, where E is energy consumed, P is Power and δt the period of time. Nevertheless, we can not make this statement with 100% confidence due to the results of the Pearson and Spearman Correlation Coefficient tests which showed p-values close to 1 as well as close to 0 for the relationship between runtime and package energy consumption for the same version and different benchmarks. An inverse relationship between runtime and energy consumption can reinforce the abovementioned uncertainty due to some cases, such as version 3.7 of C++ in the Fasta Benchmark, where runtime increases, but the energy consumption decreases.

As for the other relationships between the gathered metrics, something similar to the abovementioned connections can be observed, with p-values varying wildly for the same version and different benchmarks. Except for DRAM energy consumption and runtime, which suggests there isn't a correlation between these two metrics, but once again, we can not be 100% confident when stating this for C and Java due to the existence of quite a few outliers, like the results obtained in the Spearman test in K-Nucleotide for Java, Binary Trees for C. For Python and C++, we can say it with more confidence as there are substantially fewer significant outliers than the other two.

Another interesting point is that the peak memory usage throughout all versions of C and C++ remains relatively unchanged, only changing by insignificant amounts. The same can not be said for Java and Python, as the former seems to use less memory on the older versions than the most recent ones, and the latter appears to be more random in terms of evolution as, in some cases appear to be increasing the peak memory usage, others decreasing and others stabilizing.

With the previous behavior for peak memory usage, it is challenging to see if any correlation exists between it, energy consumption, and runtime. Regardless, the correlation tests show a solid

relationship between these metrics in many cases, and in others, it seems insignificant. Hence, we can not make any conclusions regarding these correlations.

As we can observe, reaching any conclusion related to the correlation between any metric seems nigh impossible, as both correlation tests can give wildly different values for the same version but for other benchmarks. These diversified results could have multiple causes, such as issues with the compiler or program causing performance issues, faulty libraries, issues with the test machine, or even issues when performing the correlation coefficient tests. For instance, the Pearson test has trouble with repeating values as it violates its principle of independent variables [21]. The reality is that a project like this has many variables we can not control, and managing multiple versions of the same compilers/interpreters is no easy feat requiring hours to set up and test them without the guarantee that everything is working as intended. Therefore, no conclusion can be made about the relationships between these metrics.

Regarding **RQ4: How can we rank the versions for each programming language according to the gathered data? Moreover, if so, what are the results?**, as previously mentioned, we used the ranking function to attribute a score and rank according to the performance of each benchmark used. These general rankings show that only Java shows unmistakable energy, and runtime performance increases with each release. As for the other languages, we can see that energy and runtime performance sway from version to version but still performs better in the most recent versions. For each language, the best version according to both runtime and energy efficiency of these metrics is present in the table 4.1.

Table 4.1: Best version of each language in terms of energy efficiency and runtime

Language	**Best Version**
Java	19.0.2
C	13.0.1
C++	12.0.1
Python	3.11.2

In addition to the previous observations, we also observe that the versions following the best have worse performance than said version, except for Python, which could imply that the performance of the languages is deteriorating and could require more updates and optimizations. As for the potential causes of the abovementioned oscillations, these could be due to language maturity, experimental features, or less support by the developers for specific versions due to popularity or even adoption of said version. However, there is no particular way to know for sure what these reasons are.

As for how the peak memory usage evolves according to each version, there seems to be no pattern except for Python, whose peak memory usage increases with each iteration. There could be a lot of factors influencing this, but once again, it would be pure speculation as to why this happens. Still, the best versions for peak memory usage of each language are in the following table 4.2.

Table 4.2: Best version of each language in terms of peak energy usage

Language	**Best Version**
Java	8u202
C	4.0.0
C++	11.1.0
Python	3.4.10

In conclusion, we suspect that energy efficiency is not the priority of compiler/interpreter developers when developing new updates and releases for the compilers/interpreters. A correlation between runtime and energy consumption also seems to exist, even though correlation tests do not prove it. Most languages are decreasing their energy consumption. However, their performance sways a bit from version to version as well as versions after the best version seems to have worse performance. Finally, we also show each language's best versions in terms of time and energy efficiency and the best versions for peak memory consumption.

Chapter 5

Conclusion

The theme of this dissertation is energy efficiency, so we began by evaluating the current state of compiler/interpreter version classification by energy efficiency. After said research, we present a methodology that allows us to pick the test programs, the programming languages, the respective versions of the compilers/interpreters, the tools used for data collection, how data is aggregated and cleaned, and a framework developed for this project.

With the methodology defined, we performed rigorous tests using the languages, versions, and benchmarks selected to gather the necessary information to formulate a conclusion about the current state and evolution of C, C++, Java, and Python.

We demonstrated the most energy-efficient versions for each of the languages studied, using a ranking system based on the performance of each version in the CLBG test programs.

We found a slight tendency towards lower energy consumption from the results. However, the most recent versions after the best usually have a worse performance which could imply a deteriorating performance. We also suspect that compiler/interpreter programmers do not consider energy efficiency when developing updates due to the existence of oscillation in the results of the programs.

Finally, we also tried to map the potential causes of decreases and increases in energy efficiency, but it was impossible to reach a meaningful conclusion with 100% confidence.

Despite the difficulties in finding the causes of increases and decreases in energy consumption, it is possible to make any current or future project more energy-efficient by using this dissertation as a guide to select the most appropriate language to develop a project, as well as the best compiler/interpreter version to achieve the lowest power consumption.

This dissertation is thus of great value in terms of sustainability.

As for possible future work, we could expand the number of languages to test and include total memory usage in the metrics collected. For clarification, this dissertation did not collect total memory usage due to time constraints, as the data collection took more than one week for only three benchmark programs.

5.1 Acknowledgments

I would like to thank Bernardo Santos for the help provided in co-developing the test framework.

References

[1] Acpid ubuntu manuals. https://manpages.ubuntu.com/manpages/trusty/man8/acpid.8.html.

[2] Bartlett's test. https://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm.

[3] Linux man pages - kerneloops. https://linux.die.net/man/8/kerneloops.

[4] Linux man pages - kmod. https://man7.org/linux/man-pages/man8/kmod.8.html.

[5] Linux man pages - udev. https://linux.die.net/man/8/udev.

[6] Rapl: Running average power limit. https://powerapi-ng.github.io/rapl.html.

[7] Results google sheet. https://bit.ly/compverefficency.

[8] Test framework github page. https://github.com/bernas670/resource-probe.

[9] Tiobe: Programming languages popularity ranking website. https://www.tiobe.com/tiobe-index/.

[10] Ubuntu community help wiki - cron. https://help.ubuntu.com/community/CronHowto.

[11] Ubuntu manuals - dbus-daemon. https://manpages.ubuntu.com/manpages/trusty/man1/dbus-daemon.1.html.

[12] Ubuntu manuals - gdm. https://manpages.ubuntu.com/manpages/trusty/man8/gdm.8.html.

[13] Ubuntu manuals - irqbalance. https://manpages.ubuntu.com/manpages/focal/man1/irqbalance.1.html.

[14] Ubuntu security - apparmor. https://ubuntu.com/server/docs/security-apparmor.

[15] Ubuntu wiki - apport. https://wiki.ubuntu.com/Apport.

[16] Ubuntu wiki - plymouth. https://wiki.ubuntu.com/Plymouth.

[17] Unix time tool. https://man7.org/linux/man-pages/man1/time.1.html.

[18] Which programming language is fastest? (benchmarks game). https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[19] Correlation (pearson,kendall,spearman), Aug 2021. https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/correlation-pearson-kendall-spearman/.

[20] Sarah Abdulsalam, Donna Lakomski, Qijun Gu, Tongdan Jin, and Ziliang Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*, pages 1–6. IEEE, 2014.

[21] Jonathan Z Bakdash and Laura R Marusich. Repeated measures correlation. *Frontiers in psychology*, 8:456, 2017.

[22] Xinbo Chen and Ziliang Zong. Android app energy efficiency: The impact of language, runtime, compiler, and implementation. In *2016 IEEE international conferences on big data and cloud computing (BDCloud), social computing and networking (socialcom), sustainable computing and communications (sustaincom)(BDCloud-socialcom-sustaincom)*, pages 485–492. IEEE, 2016.

[23] Ralph B D'agostino, Albert Belanger, and Ralph B D'Agostino Jr. A suggestion for using powerful and informative tests of normality. *The American Statistician*, 44(4):316–321, 1990.

[24] Kalyanmoy Deb and Himanshu Gupta. Searching for robust pareto-optimal solutions in multi-objective optimization. In *Evolutionary Multi-Criterion Optimization: Third International Conference, EMO 2005, Guanajuato, Mexico, March 9-11, 2005. Proceedings 3*, pages 150–164. Springer, 2005.

[25] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. Analyzing programming languages' energy consumption: An empirical study. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, pages 1–6. Universidade do Minho, 2017.

[26] Kamil Halbiniak, Roman Wyrzykowski, Lukasz Szustak, Adam Kulawik, Norbert Meyer, and Pawel Gepner. Performance exploration of various c/c++ compilers for amd epyc processors in numerical modeling of solidification. *Advances in Engineering Software*, 166:103078, 2022.

[27] Zofia Hanusz and Joanna Tarasińska. Normalization of the kolmogorov–smirnov and shapiro–wilk tests of normality. *Biometrical Letters*, 52(2):85–93, 2015.

[28] Peng HongYu, Sun FuJian, Wang Kan, Hao TianLu, Xiao DeQuan, and Xu LeXi. A non-cooperative data center energy consumption optimization strategy based on sdn structure. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1386–1390. IEEE, 2021.

[29] Pei Huang, Benedetta Copertaro, Xingxing Zhang, Jingchun Shen, Isabelle Löfgren, Mats Rönnelid, Jan Fahlen, Dan Andersson, and Mikael Svanfeldt. A review of data centers as prosumers in district energy systems: Renewable energy integration and waste heat reuse for district heating. *Applied energy*, 258:114109, 2020.

[30] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1, 2007.

[31] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[32] Lukas Koedijk and Ana Oprescu. Finding significant differences in the energy consumption when comparing programming languages and programs. In *2022 International Conference on ICT for Sustainability (ICT4S)*, pages 1–12. IEEE, 2022.

[33] Yanan Liu, Xiaoxia Wei, Jinyu Xiao, Zhijie Liu, Yang Xu, and Yun Tian. Energy consumption and emission mitigation prediction based on data center traffic and pue for global data centers. *Global Energy Interconnection*, 3(3):272–282, 2020.

[34] Samuel B Lyerly. The average spearman rank correlation coefficient. *Psychometrika*, 17(4):421–428, 1952.

[35] Daniel Maia, Marco Couto, João Saraiva, and Rui Pereira. E-debitum: managing software energy debt. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pages 170–177, 2020.

[36] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[37] Lloyd S Nelson. The anderson-darling test for normality. *Journal of Quality Technology*, 30(3):298, 1998.

[38] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, 2017.

[39] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Elsevier*, 205:102609, 2021.

[40] Fritz W Scholz and Michael A Stephens. K-sample anderson–darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987.

[41] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[42] Jiangjiang Wang, Hongda Deng, Yi Liu, Zeqing Guo, and Yongzhen Wang. Coordinated optimal scheduling of integrated energy system for data center based on computing load shifting. *Energy*, 267:126585, 2023.