On the Energy Efficiency of Sorting Algorithms

Francisco Neves
Universidade do Minho
Portugal
pg50375@alunos.uminho.pt

Gabriela Prata
Universidade do Minho
Portugal
pg50390@alunos.uminho.pt

Miguel Martins
Universidade do Minho
Portugal
pg50655@alunos.uminho.pt

Abstract

In the context of ES in MSc at the University of Minho, we were asked to check the different performances of languages and sorting algorithms. To do so, we monitored and measured multiple executions of 8 implementations of sorting algorithms generated by GPT-3.5 [4] in 8 different languages with three arrays of different sizes. Our results show multiple findings, such as slower/faster languages and sorting algorithms, consuming less/more energy languages and sorting algorithms, and consuming less/more memory languages and sorting algorithms.

Keywords Green Software, Programming Languages, Sorting Algorithms, GPT-3.5

1 Introduction

Nowadays, a plethora of programming languages are used in various contexts. Interpreted or compiled, or even amongst different programming paradigms, they are all at the disposal of the developers. There is a growing concern with multiple factors, such as the safety of the programs we develop, their efficiency or, sometimes, the difficulty in development. All will mostly depend on the project's context, motivations and goals.

Even if the energetic preoccupations arose primarily in the hardware industry, they are also a preoccupation to current developers [26] since we are leading with multiple ambient concerns and the necessity to make our devices more and more portable. Thus, it is necessary to be aware of the energetic efficiency of our programs. Will the language or the algorithms utilized have any impact on this topic? What are the most energetically efficient languages?

There is an extensive, although recent, research area regarding these concerns. Moreover, there are multiple techniques of optimization and analysis already developed. Amongst many, some already provided knowledge on the energy efficiency of data structures [17, 25] and Android applications [22], the energy impact of different programming practices in mobile [18, 19, 21] and desktop [23, 29], the efficiency of applications within the same scope [10, 11], the impact of the chosen programming language in energy efficiency [24] and even on how to predict the energy consumption in systems [13, 16].

ES'23, 2023
2023.

While researching, multiple questions like *if a faster program is also an energy-efficient program* or *if a faster language is a greener one* or even *if a less memory-consuming program also consumes less energy* often arise. Many of these questions aren't as linear to answer as they may seem since the energy consumption of a program will depend on multiple variables such as *Power* or *Execution Time*. There are already studies showing different results to some of these usual questions [9, 20, 25, 27, 30, 31].

In this paper, we aim to study these questions while providing data to help the reader choose a programming language for a project according to its needs. First, we start by generating eight sorting algorithms in eight programming languages, with the help of the arising new technologies and prompt engineering, using GPT-3.5 [4]. Then, we manually check the code regarding its faithfulness to the pseudocode for that sorting algorithm, which is already well established. After executing the algorithms with multiple arrays of different sizes, we aim to generate a ranking of sorting algorithms and languages. At the same time, we also analyze the computational costs (energy and time) of compiling the different programs in the chosen compiled languages.

This work builds on previous studies [14, 24], which present frameworks that allow the monitoring of the multiple evaluated metrics in the various chosen programming languages and sorting algorithms. We used these studies to analyze energy efficiency, execution time and peak memory usage in 8 programming languages with 8 algorithms each, providing 64 different algorithms to be analyzed.

The following paper is structured as follows: Section 2 explains the steps taken to achieve the desired results, and Section 3 shows the obtained results. Section 4 contains the analysis and discussion of the obtained results and the developed ranking. Section 5 presents the validity of our study. Section 6 shows the related work, and Section 7 presents the conclusions of our research.

2 Methodology

In order to achieve the desired results, we have taken multiple steps that will be described in the next subsections.

2.1 Implementations of Sorting Algorithms

We started by choosing the algorithms and the languages we would be using in our monitoring.

At first, we attempted to use all the languages listed in the "The Computer Language 23.03 Benchmarks Game" [8], but

that was not feasible since the time we had to do our research and the physical hardware were limited. There were other problems that we couldn't overcome, which made us remove languages from our list, such as Java or Kotlin, which have a limit on the source code size, leading to us not being able to declare our arrays statically.

We followed almost the same process regarding the sorting algorithms: starting our research with all the algorithms in an online list, [7] with its complexity analysis, stability and used methods. We had to discard some of the algorithms because GPT-3.5 [4] didn't generate correct implementations in the desired languages and due to our time limitations.

That way, we continued our research with 8 languages, with the same 8 sorting algorithms. To evaluate multiple use case scenarios, we used 3 different input arrays with 25 000 elements, 50 000 elements and 100 000 elements each. These were generated in a Python script with the `random` [5] library with seed 10.

The languages used were chosen because of their differences. This allowed us to compare different kinds of programming patterns and languages and compare modern languages, like Go and Rust, with older ones, like C.

The sorting algorithms were chosen based on their usage by the community and GPT-3.5 [4]'s ability to generate them.

We used the following languages:

- C;
- C++;
- C#;
- Go;
- Haskell;
- Python;
- Rust;
- Swift.

And the following sorting algorithms:

- Bubble Sort;
- Cycle Sort;
- Heap Sort;
- Insertion Sort;
- Merge Sort;
- Odd-Even Sort;
- Quick Sort;
- Selection Sort.

2.2 Energy Consumption Monitorization

To measure the energy consumption of our programs, we utilized a version we adapted from the developed framework of Intel's Running Average Power Limit (RAPL) [1] in the "Energy Efficiency across Programming Languages" [24], which is capable of providing accurate energy estimates at a very fine-grained level with an insignificant overhead, as it has already been proved [15, 24, 28].

The original implementation showed some lacking scientific methods in order to start the processes, using an arbitrary `sleep` instead of a scientific methodology. To improve that system, we used the library `lm-sensors` to get the CPU temperature and waited until the CPU got to a stable temperature (60ºC in our testing environment) to start the following program.

```
for (i = 0; i < ntimes; i++) {
    while(current_temperatue >= MIN_TEMPERATURE) {
        // get current temperature and update it
    }
    // execute algorithm and get rapl values
}
```

Since our CPU is from a modern Intel architecture, we weren't able to get values for the DRAM column, so we removed it from the obtained dataset.

We also used the library `sys/resource` [2] to get the peak memory usage of our programs, allowing us to study the associations between the execution time, energy usage and memory usage of the multiple algorithms and languages.

This way, we call the program we want to run as a child process to measure only the memory it will consume. Another way of doing this would be with the `time` UNIX command, but with this approach, we allow this data to be collected inside the main program efficiently.

```
#ifdef MEMORY_USAGE
    struct rusage rusage;
    long peakMemory = 0;

    // Execute the system command and retrieve resource
     usage
    if (system(command) != -1) {
      if (getrusage(RUSAGE_CHILDREN, &rusage) != -1) {
        peakMemory = rusage.ru_maxrss;
      }
    }
#else
    system(command);
#endif
```

2.3 Limiting the Energy Consumption with PowerCap

To check the difference the energy makes in the studied algorithms, we used the library `raplcap` [6] to limit the work rate allowed to our CPU.

This way, we reproduced our studies in a version without this cap and a version with a cap of 20 watts to our CPU work rate. The chosen value is due to the normal consumption of our CPU being around 45 watts and, this way, we reduce it around 50% to check if it makes any difference.

```
#ifdef POWERCAP
  raplcap rc = powercap(); // activates powercap
  show_power_limit(core);
#endif

for (i = 0; i < ntimes; i++) {
  // execute algorithm and get rapl values
}

#ifdef POWERCAP
```

```
  destroy_raplcap(rc); // disables powercap
#endif
```

2.4 Design and Execution

To get our results, we have the following programs in the multiple chosen languages:

Program	Description	Input
bubblesort	Bubble Sort algorithm	25000
bubblesort	Bubble Sort algorithm	50000
bubblesort	Bubble Sort algorithm	100000
cyclesort	Cycle Sort algorithm	25000
cyclesort	Cycle Sort algorithm	50000
cyclesort	Cycle Sort algorithm	100000
heapsort	Heap Sort algorithm	25000
heapsort	Heap Sort algorithm	50000
heapsort	Heap Sort algorithm	100000
insertionsort	Insertion Sort algorithm	25000
insertionsort	Insertion Sort algorithm	50000
insertionsort	Insertion Sort algorithm	100000
mergesort	Merge Sort algorithm	25000
mergesort	Merge Sort algorithm	50000
mergesort	Merge Sort algorithm	100000
oddevensort	Odd-Even Sort algorithm	25000
oddevensort	Odd-Even Sort algorithm	50000
oddevensort	Odd-Even Sort algorithm	100000
quicksort	Quick Sort algorithm	25000
quicksort	Quick Sort algorithm	50000
quicksort	Quick Sort algorithm	100000
selectionsort	Selection Sort algorithm	25000
selectionsort	Selection Sort algorithm	50000
selectionsort	Selection Sort algorithm	100000

Table 1. Programs to monitor

We can also sort our languages by paradigm and by if they are compiled, run in VMs or interpreted. In languages that have both options, we choose to utilize the compiled versions because they would be more similar to the software in production of those languages.

In order to compile our programs, the following commands were utilized:

The following compiler versions were used:

All experiments were executed five times and the values generated by RAPL were saved in a CSV file. The executions with overflows were removed from that file.

We conduct all of our research in a notebook with the following specifications: Arch Linux x86_64 operating system, kernel version 6.3.2-arch1-1, with 16GB of RAM and a Kaby Lake Intel i7-8750H (12) CPU @ 4.100GHz.

Paradigm	Languages
Functional	Haskell, Rust
Imperative	C, C++, Go, Rust
Object-Oriented	C++, C#, Rust, Swift
Scripting	Python

Table 2. Languages sorted by paradigm

Type	Languages
Compiled	C, C++, Go, Haskell, Rust, Swift
VMs	C#
Interpreted	Python

Table 3. Languages sorted by compiled, VMs or interpreted

Language	Command
C	`gcc -O2 -o`
C++	`g++ -O2 -o`
C#	`mcs -optimize+`
Go	`go build`
Haskell	`ghc -O2 –make`
Python	-
Rust	`rustc -C opt-level=2`
Swift	`swiftc -O -whole-module-optimization`

Table 4. Compiling commands

Compiler	Version
`gcc`	13.1.1 20230429
`g++`	13.1.1 20230429
`mcs`	6.12.0.0
`go`	go1.20.4 linux/amd64
`ghc`	9.2.5
`rustc`	1.64.0 (a55dd71d5 2022-09-19)
`swiftc`	5.7.3 (swift-5.7.3-RELEASE)

Table 5. Compiler versions

3 Benchmarking the Performance of the Implementation of Sorting Algorithms

This section will be divided, firstly into the obtained results for the program executions and compiling data, and inside that division, there will be the data for both implementations with powercap and without it.

This part only aims to show the obtained results, and the next ones will focus more on the analysis of those same values.

When representing plots relative mainly to algorithms, we will only show the obtained results for the implementations of `bubblesort`, `cyclesort` and `heapsort`, since they are the first three ordered alphabetically. This is due to the size of the obtained data.

We will also restrict the quantity of plots present here, due to its quantity being too large to fit in this paper.

To consult the complete data, we recommend the reader to visit the GitHub repository and our website, to see reactive and specific plots for each language and all the obtained results.

3.1 Programs

Every program was executed 5 times in the same conditions to obtain the desired results.

3.1.1 Without PowerCap

In the tables 6, 7 and 8 we can check the obtained average values of running our algorithms without PowerCap [6] being applied.

The plots 1, 2 and 3 allow us to check the difference between each language using our algorithms in terms of energy consumption and execution time. With these charts, we can, easily check that most values are very near to each other, allowing us to create some species of clusters in order to distinguish the most efficient and less efficient languages.

In the bar plots 4, 5 and 6, we can check the difference between each language in these algorithms in terms of energy consumption and time, being able to check if there is some relation between these variables. The left axis represents the energetic consumption in joules and the right one represents the time in seconds. The bars are relative to the energy and the line to execution time.

Next, we have bar plots 7, 8 and 9, these similarly to the previous ones, allow us to take conclusions trying to relate two variables, in these case, the energetic consumption and the peak memory usage. The left axis represents the energetic consumption in joules and the right one the memory peak usage in megabytes. The bars are relative to the energy and the line to execution time.

Table 6. Average performance of Bubble Sort

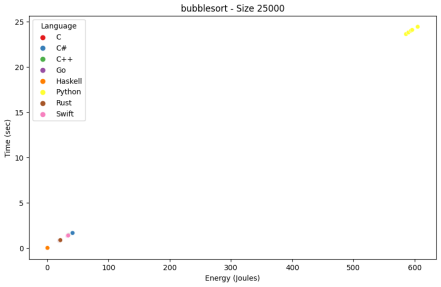
Language	Energy (J)	Memory (MB)	Time (sec)
BubbleSort 25000 input			
C	33.958	3.61	1.402
C#	40.812	20.627	1.654
C++	34.307	3.912	1.425
Go	20.074	3.635	0.817
Haskell	0.169	6.31	0.007
Python	594.055	32.611	23.989
Rust	21.055	3.648	0.858
Swift	33.428	10.716	1.353
BubbleSort 50000 input			
C	139.434	3.596	5.775
C#	167.202	21.79	6.785
C++	141.736	4.096	5.851
Go	85.274	5.484	3.47
Haskell	0.17	9.164	0.009
Python	2418.632	51.53	97.641
Rust	95.95	3.532	3.91
Swift	135.87	11.046	5.499
BubbleSort 100000 input			
C	563.863	3.584	23.293
C#	670.847	22.344	27.165
C++	570.408	4.344	23.556
Go	355.33	9.864	14.446
Haskell	0.264	16.473	0.014
Python	9904.623	93.7	399.894
Rust	395.39	3.597	16.078
Swift	550.022	11.753	22.244

Table 7. Average Performance of Cycle Sort

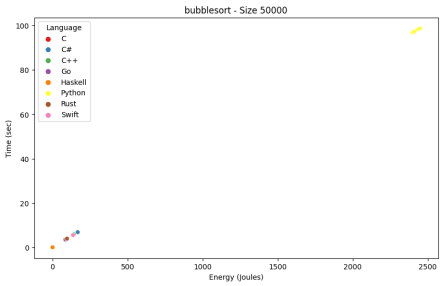
Language	Energy (J)	Memory (MB)	Time (sec)
CycleSort 25000 input			
C	34.737	3.686	1.432
C#	61.852	20.66	2.516
C++	37.397	3.869	1.556
Go	34.13	3.456	1.397
Haskell	230.714	11.504	9.318
Python	888.551	32.804	35.866
Rust	5.784	3.635	0.234
Swift	40.942	10.656	1.666
CycleSort 50000 input			
C	138.709	3.635	5.737
C#	244.367	20.718	9.943
C++	147.737	4.154	6.156
Go	137.715	3.622	5.63
Haskell	1062.992	14.592	42.903
Python	3661.929	51.569	147.918
Rust	22.918	3.661	0.927
Swift	165.549	11.032	6.725
CycleSort 100000 input			
C	552.765	3.661	22.872
C#	969.999	22.14	39.51
C++	590.273	4.415	24.56
Go	546.764	11.884	22.32
Haskell	5050.211	20.619	203.882
Python	15803.388	93.743	638.807
Rust	94.561	3.583	3.82
Swift	659.847	11.626	26.776

Table 8. Average Performance of Heap Sort

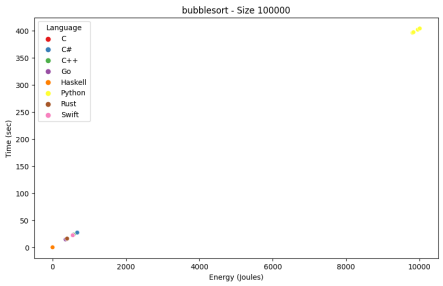
Language	Energy (J)	Memory (MB)	Time (sec)
HeapSort 25000 input			
C	0.163	3.712	0.006
C#	0.655	24.052	0.028
C++	0.178	3.898	0.007
Go	0.217	4.04	0.009
Haskell	0.106	3.699	0.004
Python	3.243	32.78	0.131
Rust	0.148	3.66	0.007
Swift	0.263	10.79	0.011
HeapSort 50000 input			
C	0.17	3.658	0.007
C#	0.932	22.154	0.046
C++	0.196	4.142	0.008
Go	0.276	3.546	0.012
Haskell	0.101	3.712	0.004
Python	6.182	51.56	0.25
Rust	0.291	3.635	0.012
Swift	0.335	11.316	0.014
HeapSort 100000 input			
C	0.273	3.648	0.013
C#	1.093	22.207	0.049
C++	0.377	4.419	0.015
Go	0.424	5.868	0.019
Haskell	0.104	3.577	0.004
Python	12.702	93.741	0.513
Rust	0.513	3.532	0.021
Swift	0.618	12.622	0.026



(a) BubbleSort 25000 energy and time

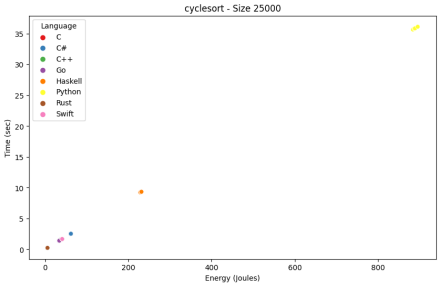


(b) BubbleSort 50000 energy and time

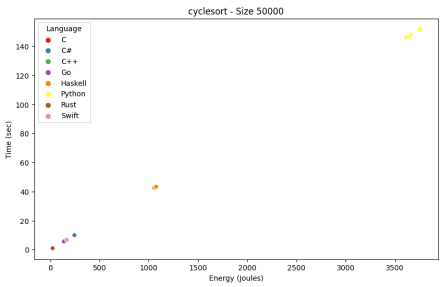


(c) BubbleSort 100000 energy and time

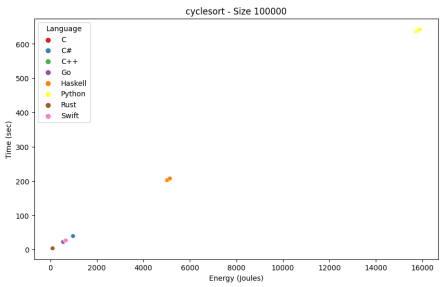
Figure 1. Bubble Sort Chart Energy and Time Program Without PowerCap



(a) CycleSort 25000 energy and time

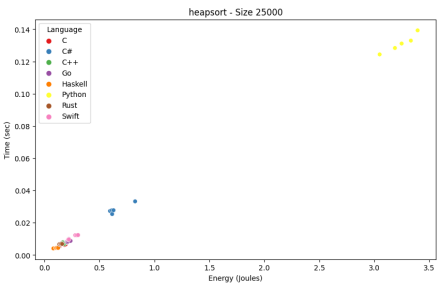


(b) CycleSort 50000 energy and time

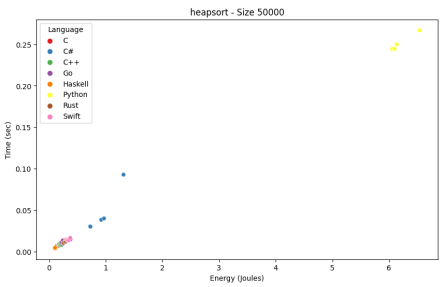


(c) CycleSort 100000 energy and time

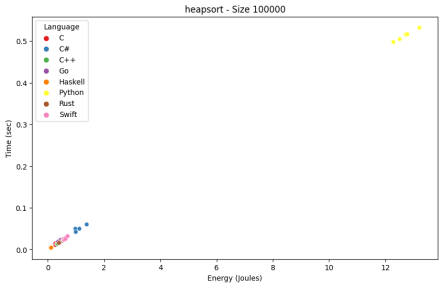
Figure 2. Cycle Sort Chart Energy and Time Program Without PowerCap



(a) HeapSort 25000 energy and time

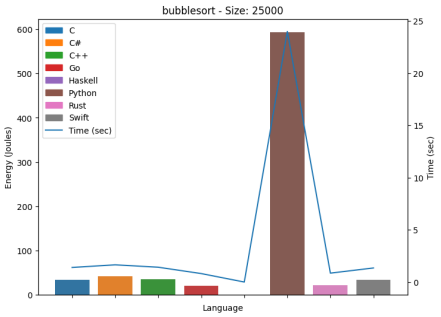


(b) HeapSort 50000 energy and time

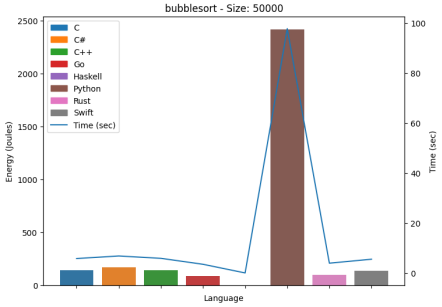


(c) HeapSort 100000 energy and time

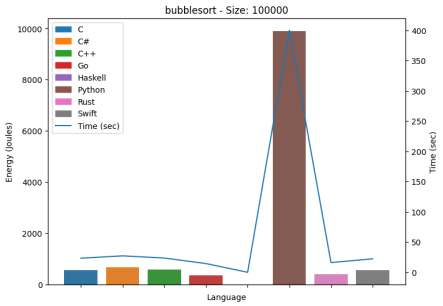
Figure 3. Heap Sort Chart Energy and Time Program Without PowerCap



(a) BubbleSort 25000 energy and time

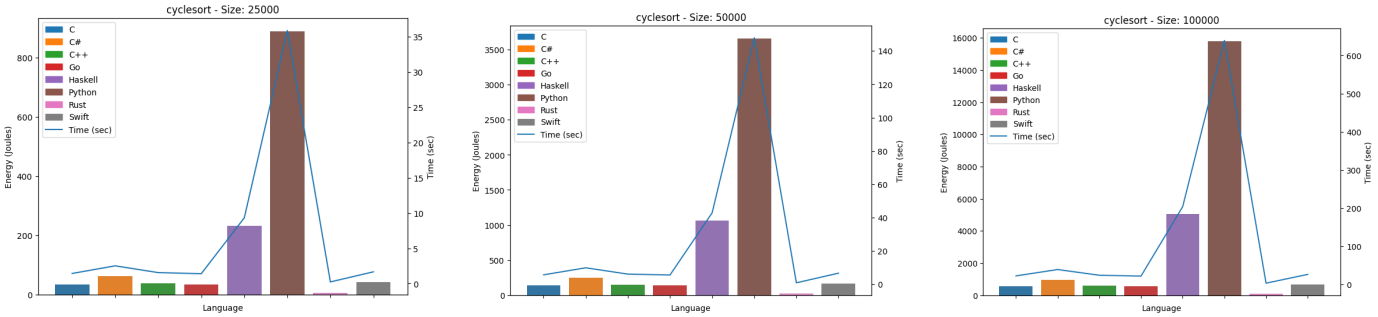


(b) BubbleSort 50000 energy and time



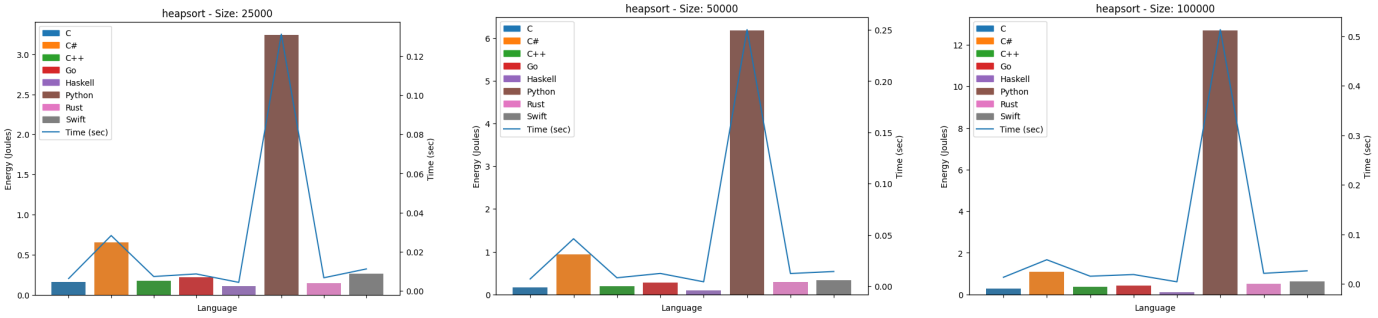
(c) BubbleSort 100000 energy and time

Figure 4. Bubble Sort Plot Bar Energy and Time Program Without PowerCap



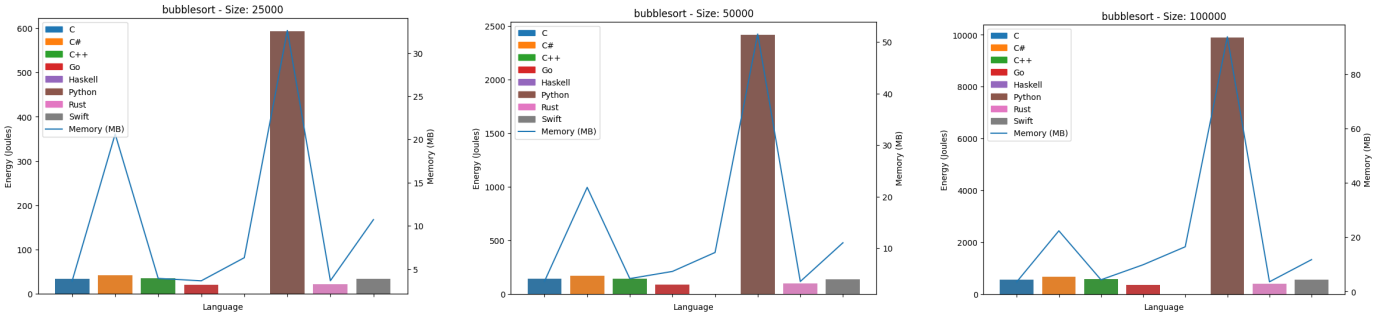
(a) CycleSort 25000 energy and time **(b)** CycleSort 50000 energy and time **(c)** CycleSort 100000 energy and time

Figure 5. Cycle Sort Plot Bar Energy and Time Program Without PowerCap



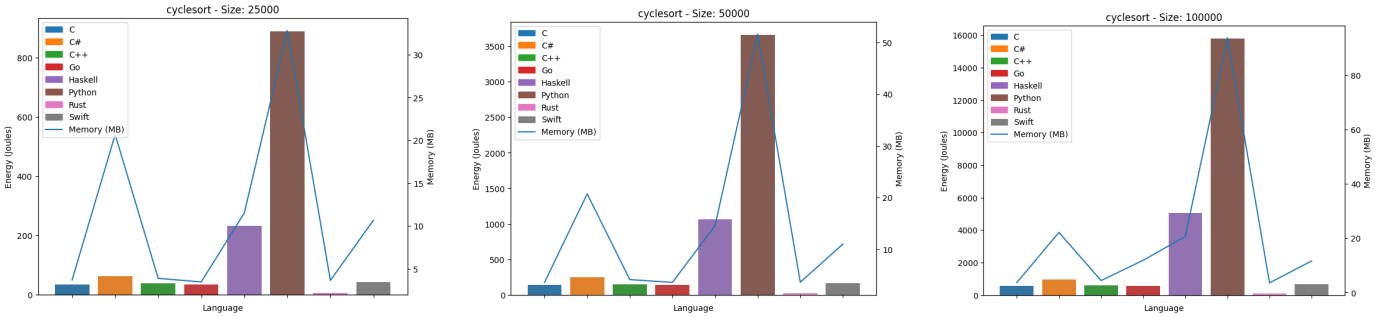
(a) HeapSort 25000 energy and time **(b)** HeapSort 50000 energy and time **(c)** HeapSort 100000 energy and time

Figure 6. Heap Sort Plot bar Energy and Time Program Without PowerCap



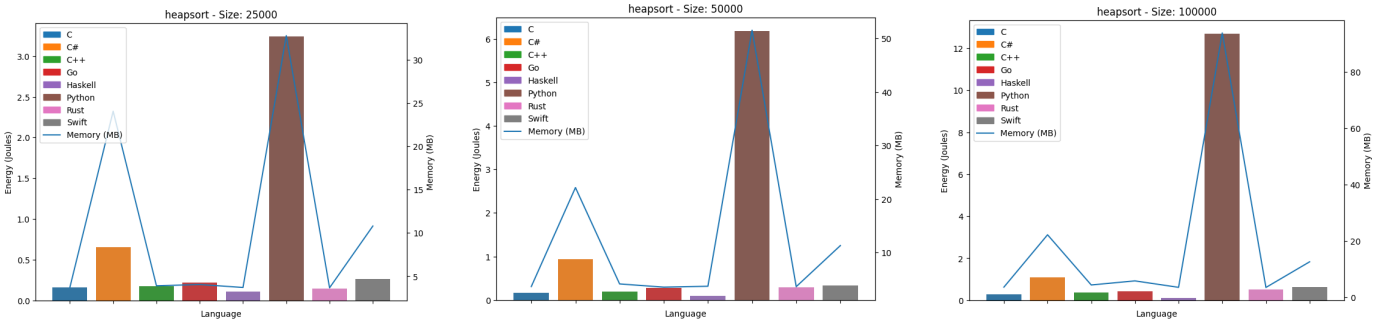
(a) BubbleSort 25000 energy and memory **(b)** BubbleSort 50000 energy and memory **(c)** BubbleSort 100000 energy and memory

Figure 7. Bubble Sort Plot Bar Energy and Memory Program Without PowerCap



(a) CycleSort 25000 energy and memory **(b)** CycleSort 50000 energy and memory **(c)** CycleSort 100000 energy and memory

Figure 8. Cycle Sort Plot Bar Energy and Memory Program Without PowerCap



(a) HeapSort 25000 energy and memory **(b)** HeapSort 50000 energy and memory **(c)** HeapSort 100000 energy and memory

Figure 9. Heap Sort Plot bar Energy and Memory Program Without PowerCap

3.1.2 With PowerCap

In order to check if any difference was made when restraining the work rate for a program, we used `raplcap` [6], a version of PowerCap. This way, in the equation of Energy ($Energy = Power \times Time$), we are restricting the Power. The chosen value to do this was 20W, once the CPU in which we've done our tests had a default maximum work rate of 45W, limiting it, this way, around 50%.

In the tables 9, 10 and 11 we can check the obtained average values of running our algorithms applying this cap.

The plots 10, 11 and 12 allow us to check the difference between each language using our algorithms in terms of energy consumption and execution time. With these charts, we can, easily check that most values are very near to each other, allowing us to create some species of clusters in order to distinguish the most efficient and less efficient languages.

In the bar plots 13, 14 and 15, we can check the difference between each language in these algorithms in terms of energy consumption and time, being able to check if there is some relation between these variables. The left axis represents the energetic consumption in joules and the right one represents the time in seconds. The bars are relative to the energy and the line to execution time.

Next, we have bar plots 16, 17 and 18, these similarly to the previous ones, allow us to take conclusions trying to relate two variables, in these case, the energetic consumption and the peak memory usage. The left axis represents the energetic consumption in joules and the right one the memory peak usage in megabytes. The bars are relative to the energy and the line to execution time.

Table 9. Average Performance of Bubble Sort with PowerCap

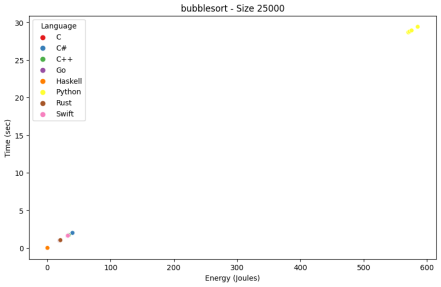
Language	Energy (J)	Memory (MB)	Time (sec)
BubbleSort 25000 input			
C	33.207	3.61	1.673
C#	39.71	22.16	1.999
C++	33.702	3.854	1.699
Go	19.546	3.712	0.985
Haskell	0.128	6.208	0.007
Python	575.256	32.568	28.899
Rust	20.466	3.648	1.032
Swift	32.092	10.567	1.615
BubbleSort 50000 input			
C	136.478	3.533	6.877
C#	162.407	21.628	8.174
C++	138.151	4.085	6.964
Go	82.971	3.507	4.176
Haskell	0.164	9.101	0.008
Python	2364.746	51.531	118.801
Rust	93.459	3.558	4.705
Swift	131.817	11.02	6.631
BubbleSort 100000 input			
C	552.013	3.597	27.828
C#	661.894	22.086	30.01
C++	556.828	4.386	28.047
Go	346.286	5.871	17.433
Haskell	0.266	16.43	0.013
Python	9744.266	93.708	489.596
Rust	383.539	3.712	19.314
Swift	537.892	11.65	27.054

Table 10. Average Performance of Cycle Sort with PowerCap

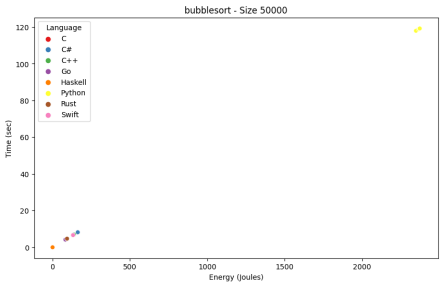
Language	Energy (J)	Memory (MB)	Time (sec)
CycleSort 25000 input			
C	33.727	3.712	1.701
C#	59.819	21.73	3.012
C++	36.414	3.802	1.837
Go	33.23	3.61	1.675
Haskell	224.044	11.494	11.249
Python	871.428	32.802	43.79
Rust	5.56	3.712	0.279
Swift	39.976	10.635	2.012
CycleSort 50000 input			
C	135.159	3.686	6.814
C#	237.847	21.122	11.976
C++	145.018	3.967	7.313
Go	133.724	3.661	6.736
Haskell	1028.143	14.515	51.632
Python	3629.614	51.602	182.391
Rust	22.109	3.546	1.111
Swift	160.788	10.984	8.096
CycleSort 100000 input			
C	539.84	3.571	27.202
C#	959.305	21.977	43.643
C++	579.062	4.413	29.204
Go	532.909	4.579	26.864
Haskell	4778.613	20.646	239.999
Python	15342.689	93.744	770.984
Rust	91.802	3.674	4.617
Swift	641.024	11.714	32.262

Table 11. Average Performance of Heap Sort with PowerCap

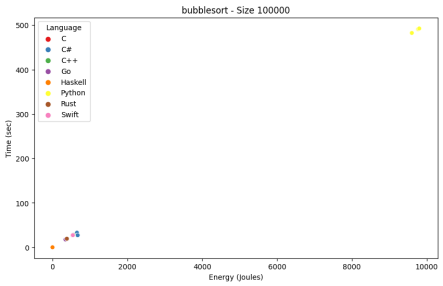
Language	Energy (J)	Memory (MB)	Time (sec)
HeapSort 25000 input			
C	0.192	3.546	0.009
C#	0.709	20.714	0.037
C++	0.197	3.885	0.01
Go	0.158	3.712	0.008
Haskell	0.106	3.571	0.006
Python	3.2	32.812	0.163
Rust	0.204	3.583	0.011
Swift	0.251	10.81	0.013
HeapSort 50000 input			
C	0.216	3.647	0.011
C#	0.777	19.886	0.04
C++	0.207	4.154	0.011
Go	0.269	3.712	0.014
Haskell	0.094	3.712	0.005
Python	5.965	51.585	0.3
Rust	0.218	3.712	0.011
Swift	0.308	11.337	0.017
HeapSort 100000 input			
C	0.272	3.533	0.015
C#	1.081	22.256	0.052
C++	0.317	4.408	0.016
Go	0.47	5.872	0.025
Haskell	0.087	3.482	0.005
Python	12.278	93.742	0.618
Rust	0.515	3.712	0.026
Swift	0.52	12.646	0.026



(a) BubbleSort 25000 energy and time

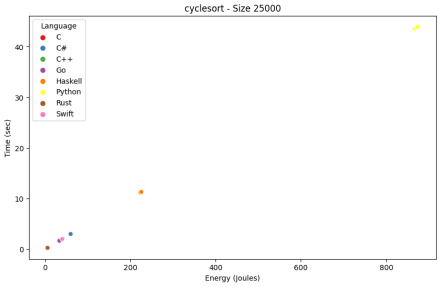


(b) BubbleSort 50000 energy and time

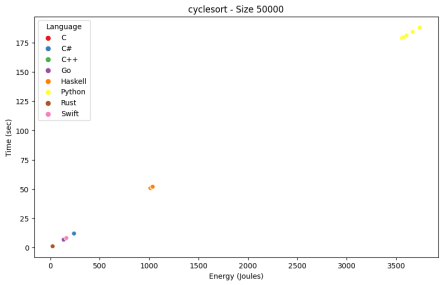


(c) BubbleSort 100000 energy and time

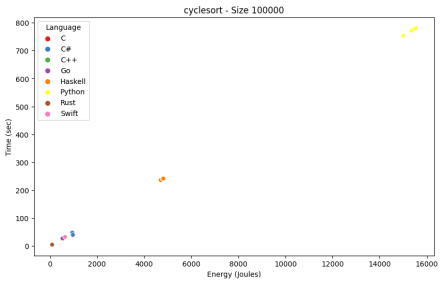
Figure 10. Bubble Sort Chart Energy and Time Program with PowerCap



(a) CycleSort 25000 energy and time

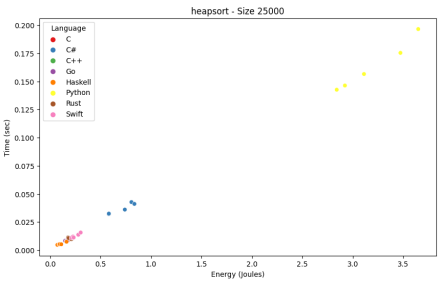


(b) CycleSort 50000 energy and time

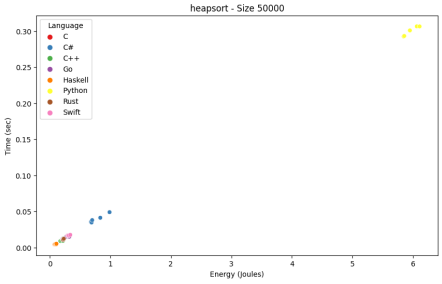


(c) CycleSort 100000 energy and time

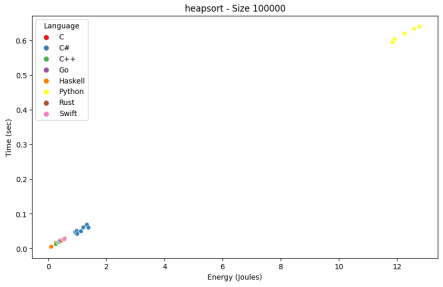
Figure 11. Cycle Sort Chart Energy and Time Program with PowerCap



(a) HeapSort 25000 energy and time

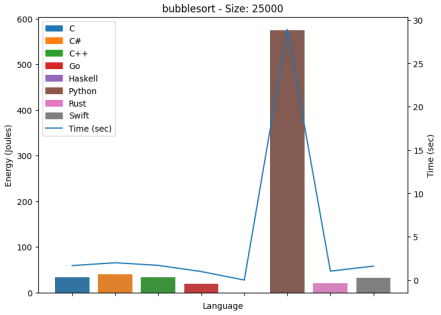


(b) HeapSort 50000 energy and time

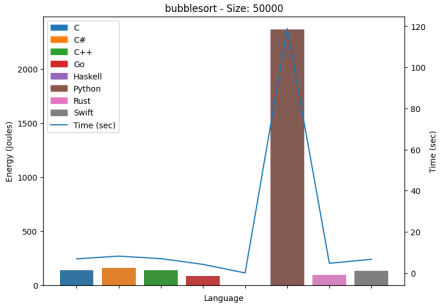


(c) HeapSort 100000 energy and time

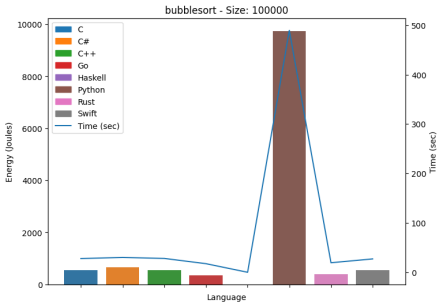
Figure 12. Heap Sort Chart Energy and Time Program with PowerCap



(a) BubbleSort 25000 energy and time

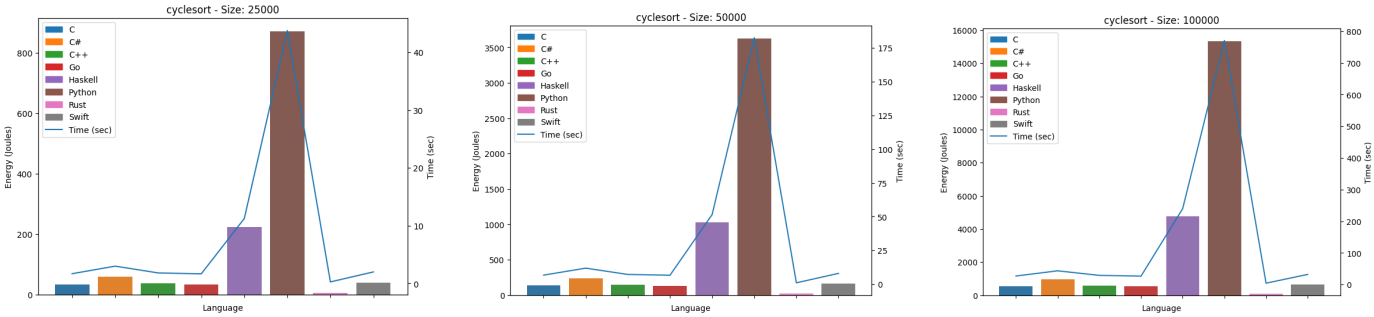


(b) BubbleSort 50000 energy and time



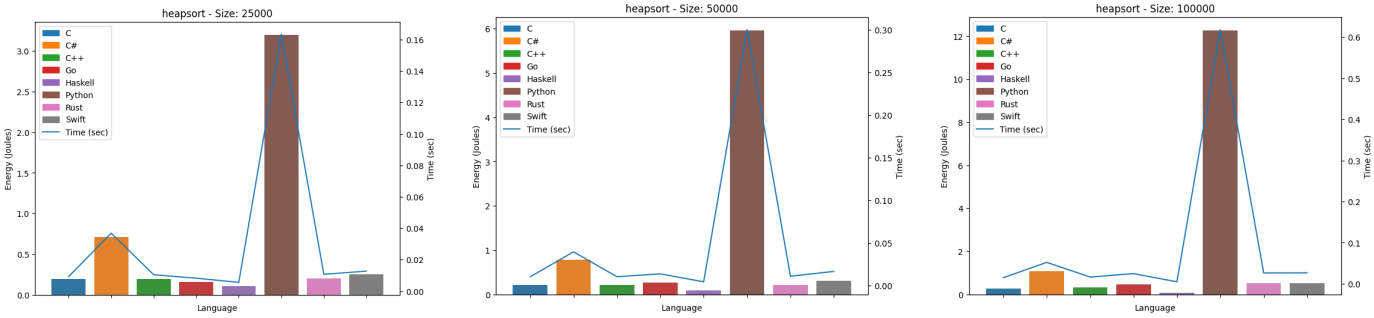
(c) BubbleSort 100000 energy and time

Figure 13. Bubble Sort Plot Bar Energy and Time Program with PowerCap



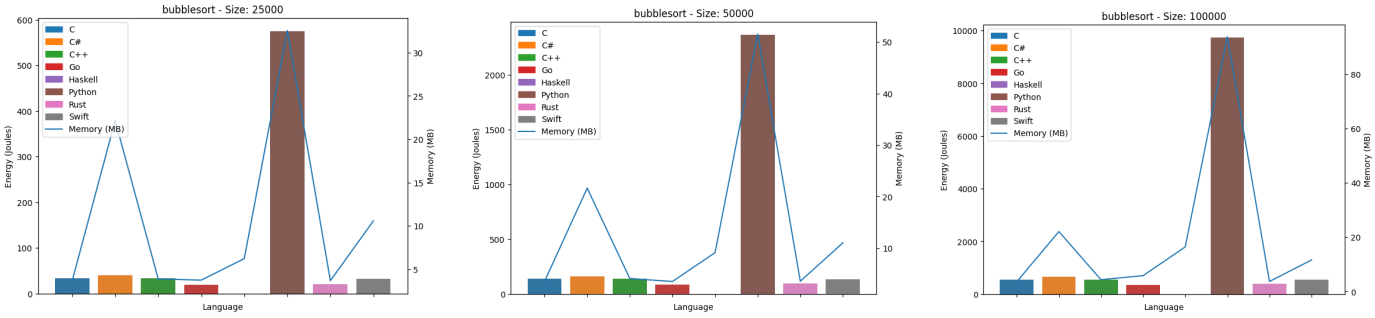
(a) CycleSort 25000 energy and time **(b)** CycleSort 50000 energy and time **(c)** CycleSort 100000 energy and time

Figure 14. Cycle Sort Plot Bar Energy and Time Program with PowerCap



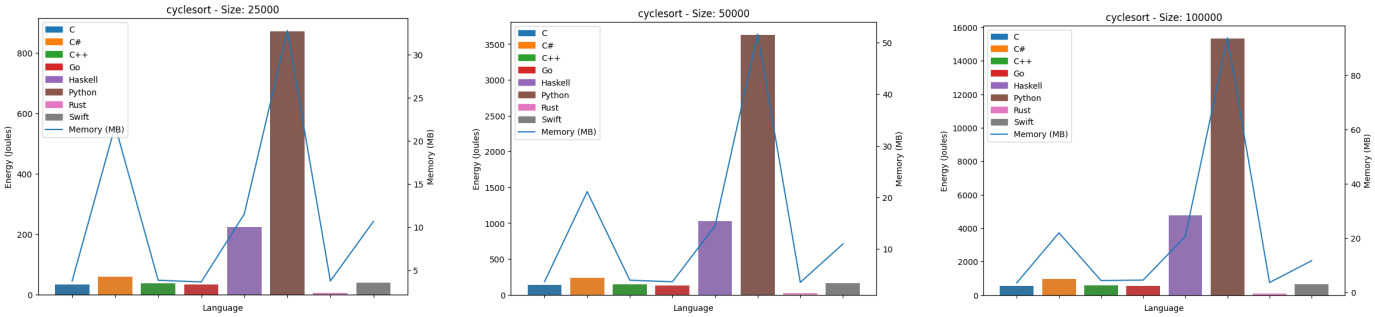
(a) HeapSort 25000 energy and time **(b)** HeapSort 50000 energy and time **(c)** HeapSort 100000 energy and time

Figure 15. Heap Sort Plot bar Energy and Time Program with PowerCap



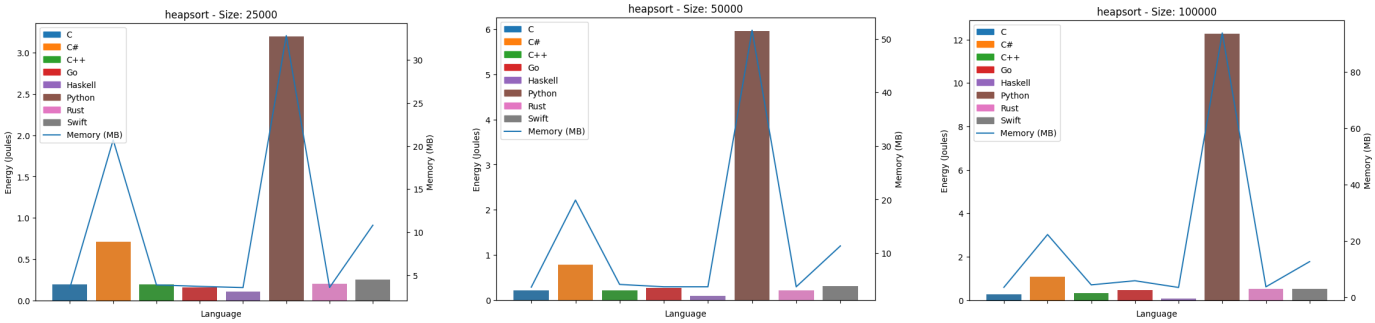
(a) BubbleSort 25000 energy and memory **(b)** BubbleSort 50000 energy and memory **(c)** BubbleSort 100000 energy and memory

Figure 16. Bubble Sort Plot Bar Energy and Memory Program with PowerCap



(a) CycleSort 25000 energy and memory **(b)** CycleSort 50000 energy and memory **(c)** CycleSort 100000 energy and memory

Figure 17. Cycle Sort Plot Bar Energy and Memory Program with PowerCap



(a) HeapSort 25000 energy and memory **(b)** HeapSort 50000 energy and memory **(c)** HeapSort 100000 energy and memory

Figure 18. Heap Sort Plot bar Energy and Memory Program with PowerCap

3.2 Compiling

Every compiling process was executed 5 times in the same conditions in order to obtain the desired results. The utilized optimizations are available on table 4 and the compiler versions on table 5.

Notice that no data is available for Python in this subsection, as Python was not compiled.

3.2.1 Without PowerCap

In the tables 12, 13 and 14 we can check the obtained average values of compiling algorithms without applying PowerCap [6].

The plots 19, 20 and 21 allow us to check the difference between each language compiling process using our algorithms in terms of energy consumption and execution time. With these charts, we can, easily check that most values are very near to each other, allowing us to create some species of clusters in order to distinguish the most efficient and less efficient compiling processes.

In the bar plots 22, 23 and 24, we can check the difference between each language compiling process in these algorithms in terms of energy consumption and time, being able to check if there is some relation between these variables. The left axis represents the energetic consumption in joules and the right one represents the time in seconds. The bars are relative to the energy and the line to execution time.

Next, we have bar plots 25, 26 and 27, these similarly to the previous ones, allow us to take conclusions trying to relate two variables, in these case, the energetic consumption and the peak memory usage. The left axis represents the energetic consumption in joules and the right one the memory peak usage in megabytes. The bars are relative to the energy and the line to execution time.

Table 12. Average Performance Compiling Bubble Sort

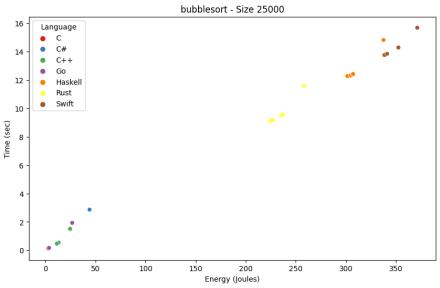
Language	Energy (J)	Memory (MB)	Time (sec)
BubbleSort 25000 input			
C	2.371	38.434	0.106
C#	18.039	65.985	0.911
C++	13.533	85.498	0.635
Go	7.152	55.888	0.444
Haskell	309.82	631.325	12.75
Rust	236.084	1082.473	9.736
Swift	347.046	143.981	14.197
BubbleSort 50000 input			
C	3.222	42.975	0.142
C#	13.388	69.291	0.543
C++	12.339	98.947	0.505
Go	3.498	57.401	0.158
Haskell	606.907	1104.613	24.578
Rust	480.337	2173.641	19.422
Swift	1195.499	220.207	48.502
BubbleSort 100000 input			
C	5.081	51.924	0.22
C#	15.07	71.479	0.615
C++	15.067	115.769	0.622
Go	3.824	57.703	0.168
Haskell	1210.423	1794.711	49.015
Rust	945.489	4323.859	38.29
Swift	316.501	366.755	13.303

Table 13. Average Performance Compiling Cycle Sort

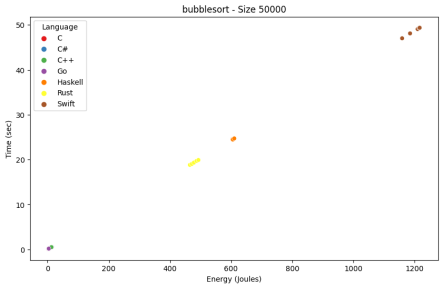
Language	Energy (J)	Memory (MB)	Time (sec)
CycleSort 25000 input			
C	2.665	38.561	0.113
C#	12.905	66.417	0.521
C++	11.573	85.687	0.471
Go	3.452	55.111	0.15
Haskell	581.574	709.001	23.546
Rust	230.529	1081.773	9.318
Swift	344.898	143.499	14.015
CycleSort 50000 input			
C	3.548	43.358	0.145
C#	13.711	69.373	0.553
C++	12.775	98.682	0.52
Go	3.379	56.716	0.152
Haskell	1175.832	1073.2	47.584
Rust	481.533	2171.166	19.471
Swift	1189.82	219.981	48.226
CycleSort 100000 input			
C	5.442	52.525	0.223
C#	15.104	71.507	0.614
C++	15.253	114.221	0.623
Go	3.553	57.423	0.172
Haskell	2422.179	2185.884	97.964
Rust	943.917	4317.879	38.189
Swift	317.451	366.659	13.338

Table 14. Average performance of heap sort

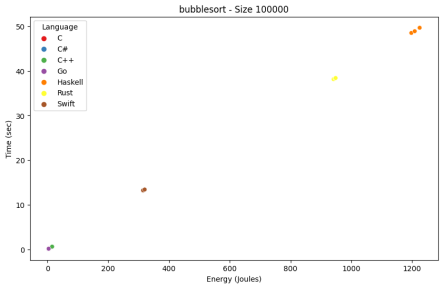
Language	Energy (J)	Memory (MB)	Time (sec)
HeapSort 25000 input			
C	2.323	37.453	0.097
C#	12.853	65.719	0.522
C++	11.57	85.423	0.471
Go	3.599	56.133	0.162
Haskell	306.608	636.957	12.418
Rust	226.749	1081.057	9.16
Swift	368.903	147.347	15.01
HeapSort 50000 input			
C	3.523	42.862	0.145
C#	13.68	66.834	0.554
C++	12.576	98.921	0.514
Go	3.682	56.327	0.161
Haskell	610.421	1036.352	24.687
Rust	471.178	2169.621	19.057
Swift	1247.474	223.376	50.6
HeapSort 100000 input			
C	5.232	51.883	0.217
C#	14.913	73.537	0.605
C++	15.343	114.527	0.624
Go	3.254	56.616	0.143
Haskell	1204.74	1740.351	48.753
Rust	938.557	4315.607	37.993
Swift	351.573	370.305	14.746



(a) BubbleSort 25000 energy and time

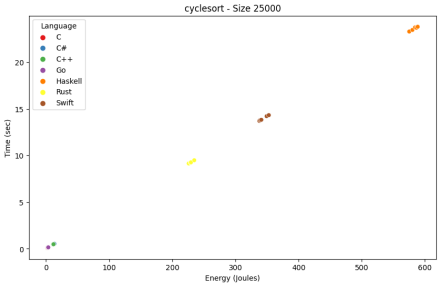


(b) BubbleSort 50000 energy and time

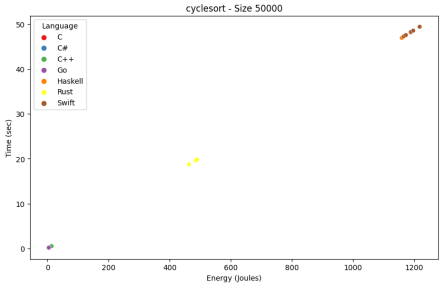


(c) BubbleSort 100000 energy and time

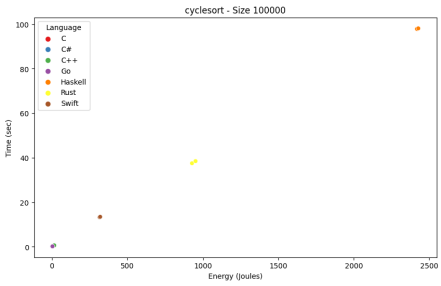
Figure 19. Bubble Sort Chart Energy and Time Compiling Without PowerCap



(a) CycleSort 25000 energy and time

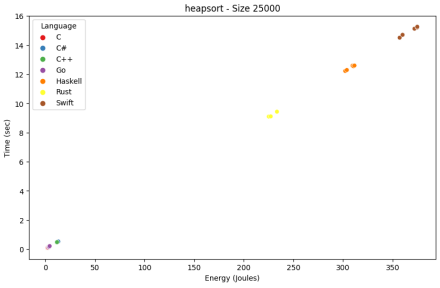


(b) CycleSort 50000 energy and time

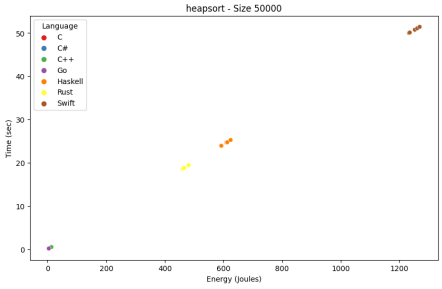


(c) CycleSort 100000 energy and time

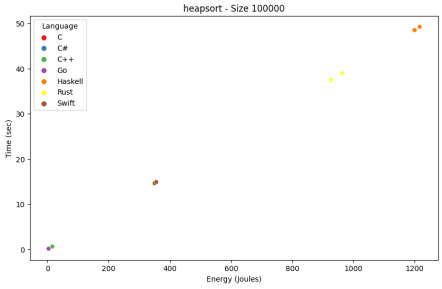
Figure 20. Cycle Sort Chart Energy and Time Compiling Without PowerCap



(a) HeapSort 25000 energy and time

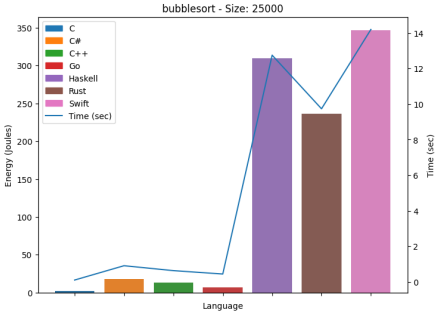


(b) HeapSort 50000 energy and time

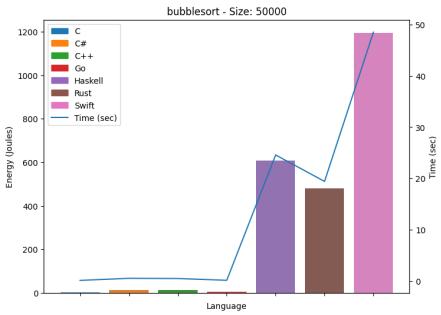


(c) HeapSort 100000 energy and time

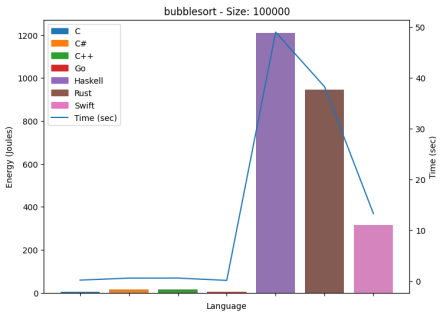
Figure 21. Heap Sort Chart Energy and Time Compiling Without PowerCap



(a) BubbleSort 25000 energy and time

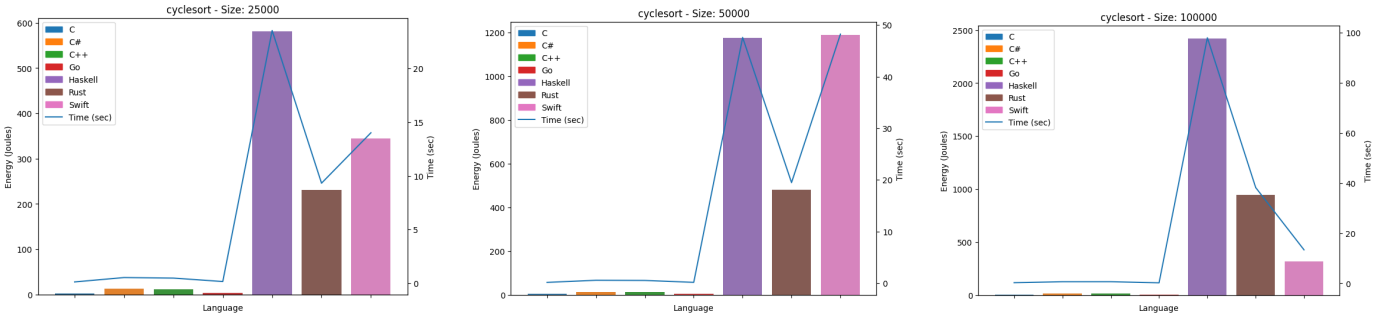


(b) BubbleSort 50000 energy and time



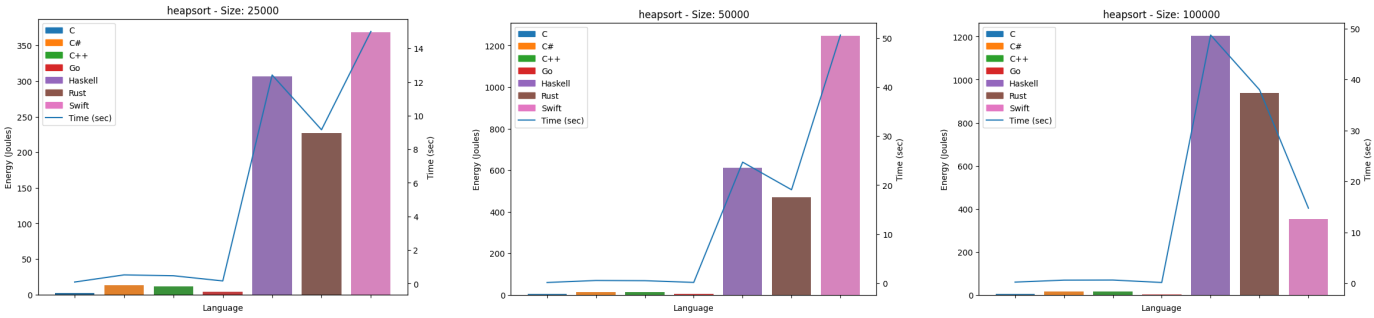
(c) BubbleSort 100000 energy and time

Figure 22. Bubble Sort Plot Bar Energy and Time Compiling Without PowerCap



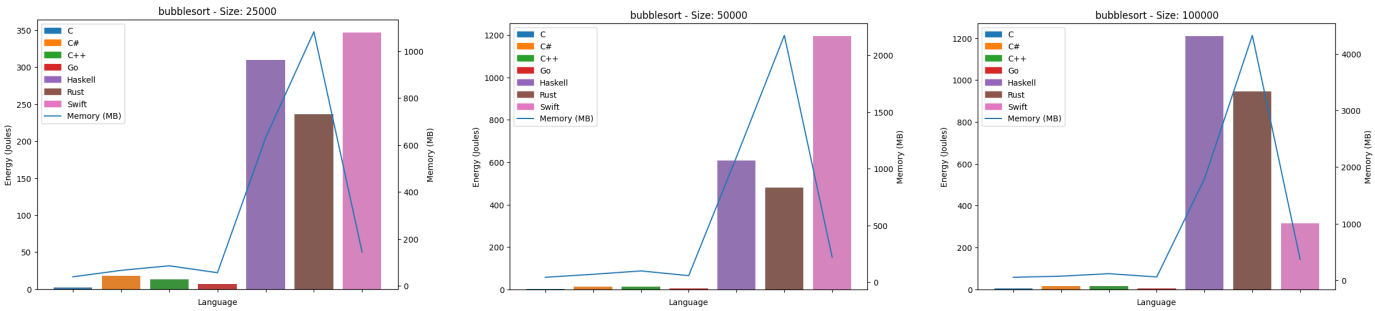
(a) CycleSort 25000 energy and time **(b)** CycleSort 50000 energy and time **(c)** CycleSort 100000 energy and time

Figure 23. Cycle Sort Plot Bar Energy and Time Compiling Without PowerCap



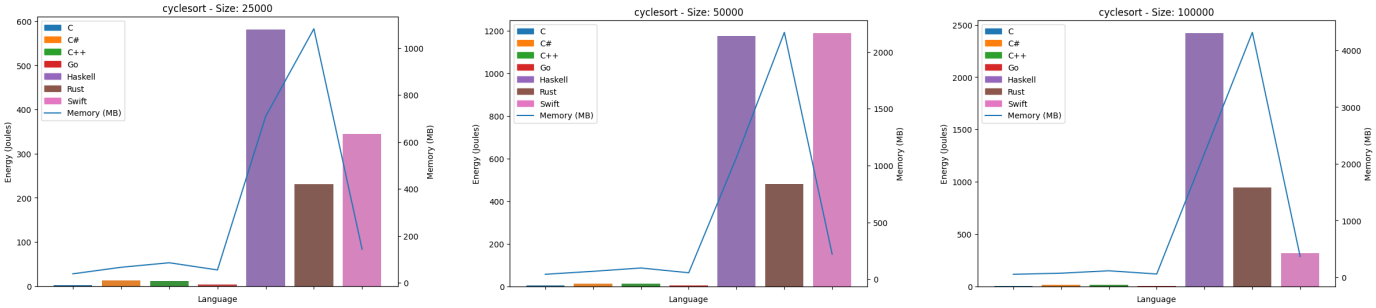
(a) HeapSort 25000 energy and time **(b)** HeapSort 50000 energy and time **(c)** HeapSort 100000 energy and time

Figure 24. Heap Sort Plot bar Energy and Time Compiling Without PowerCap



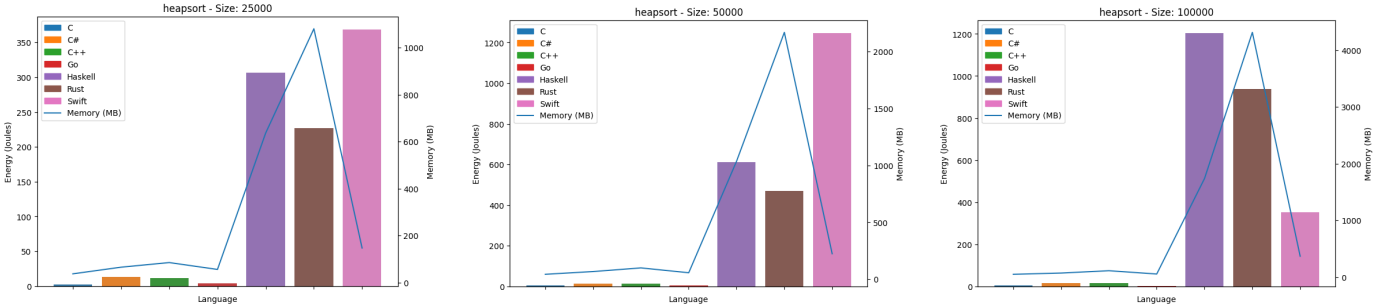
(a) BubbleSort 25000 energy and memory **(b)** BubbleSort 50000 energy and memory **(c)** BubbleSort 100000 energy and memory

Figure 25. Bubble Sort Plot Bar Energy and Memory Compiling Without PowerCap



(a) CycleSort 25000 energy and memory **(b)** CycleSort 50000 energy and memory **(c)** CycleSort 100000 energy and memory

Figure 26. Cycle Sort Plot Bar Energy and Memory Compiling Without PowerCap



(a) HeapSort 25000 energy and memory **(b)** HeapSort 50000 energy and memory **(c)** HeapSort 100000 energy and memory

Figure 27. Heap Sort Plot bar Energy and Memory Compiling Without PowerCap

3.2.2 With PowerCap

Similarly to 3.1.2, we applied a 20W PowerCap to compiling to check if there were any differences in the results.

In the tables 15, 16 and 17 we can check the obtained average values of compiling algorithms without by applying that cap.

The plots 28, 29 and 30 allow us to check the difference between each language compiling process using our algorithms in terms of energy consumption and execution time. With these charts, we can easily check that most values are very near to each other, allowing us to create some species of clusters to distinguish the most efficient and less efficient compiling processes.

In the bar plots 31, 32 and 33, we can check the difference between each language compiling process in these algorithms in terms of energy consumption and time, being able to check if there is some relation between these variables. The left axis represents the energetic consumption in joules and the right one represents the time in seconds. The bars are relative to the energy and the line to execution time.

Next, we have bar plots 34, 35 and 36, similar to the previous ones, allow us to take conclusions trying to relate two variables, in these case, the energetic consumption and the peak memory usage. The left axis represents the energetic consumption in joules and the right one the memory peak usage in megabytes. The bars are relative to the energy and the line to execution time.

Table 15. Average Performance Compiling Bubble Sort with PowerCap

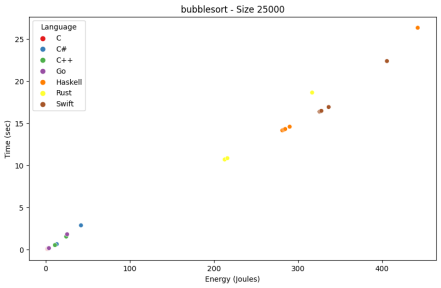
Language	Energy (J)	Memory (MB)	Time (sec)
BubbleSort 25000 input			
C	33.207	3.61	1.673
C#	39.71	22.16	1.999
C++	33.702	3.854	1.699
Go	19.546	3.712	0.985
Haskell	0.128	6.208	0.007
Python	575.256	32.568	28.899
Rust	20.466	3.648	1.032
Swift	32.092	10.567	1.615
BubbleSort 50000 input			
C	136.478	3.533	6.877
C#	162.407	21.628	8.174
C++	138.151	4.085	6.964
Go	82.971	3.507	4.176
Haskell	0.164	9.101	0.008
Python	2364.746	51.531	118.801
Rust	93.459	3.558	4.705
Swift	131.817	11.02	6.631
BubbleSort 100000 input			
C	552.013	3.597	27.828
C#	661.894	22.086	30.01
C++	556.828	4.386	28.047
Go	346.286	5.871	17.433
Haskell	0.266	16.43	0.013
Python	9744.266	93.708	489.596
Rust	383.539	3.712	19.314
Swift	537.892	11.65	27.054

Table 16. Average Performance Compiling Cycle Sort with PowerCap

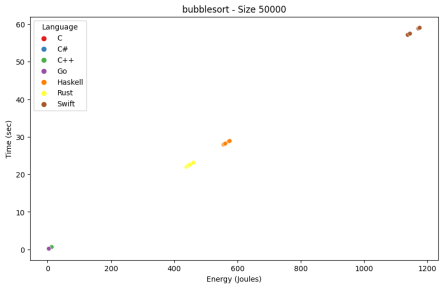
Language	Energy (J)	Memory (MB)	Time (sec)
CycleSort 25000 input			
C	33.727	3.712	1.701
C#	59.819	21.73	3.012
C++	36.414	3.802	1.837
Go	33.23	3.61	1.675
Haskell	224.044	11.494	11.249
Python	871.428	32.802	43.79
Rust	5.56	3.712	0.279
Swift	39.976	10.635	2.012
CycleSort 50000 input			
C	135.159	3.686	6.814
C#	237.847	21.122	11.976
C++	145.018	3.967	7.313
Go	133.724	3.661	6.736
Haskell	1028.143	14.515	51.632
Python	3629.614	51.602	182.391
Rust	22.109	3.546	1.111
Swift	160.788	10.984	8.096
CycleSort 100000 input			
C	539.84	3.571	27.202
C#	959.305	21.977	43.643
C++	579.062	4.413	29.204
Go	532.909	4.579	26.864
Haskell	4778.613	20.646	239.999
Python	15342.689	93.744	770.984
Rust	91.802	3.674	4.617
Swift	641.024	11.714	32.262

Table 17. Average Performance Compiling Heap Sort with PowerCap

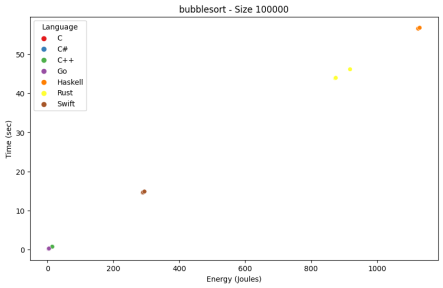
Language	Energy (J)	Memory (MB)	Time (sec)
HeapSort 25000 input			
C	0.192	3.546	0.009
C#	0.709	20.714	0.037
C++	0.197	3.885	0.01
Go	0.158	3.712	0.008
Haskell	0.106	3.571	0.006
Python	3.2	32.812	0.163
Rust	0.204	3.583	0.011
Swift	0.251	10.81	0.013
HeapSort 50000 input			
C	0.216	3.647	0.011
C#	0.777	19.886	0.04
C++	0.207	4.154	0.011
Go	0.269	3.712	0.014
Haskell	0.094	3.712	0.005
Python	5.965	51.585	0.3
Rust	0.218	3.712	0.011
Swift	0.308	11.337	0.017
HeapSort 100000 input			
C	0.272	3.533	0.015
C#	1.081	22.256	0.052
C++	0.317	4.408	0.016
Go	0.47	5.872	0.025
Haskell	0.087	3.482	0.005
Python	12.278	93.742	0.618
Rust	0.515	3.712	0.026
Swift	0.52	12.646	0.026



(a) BubbleSort 25000 energy and time

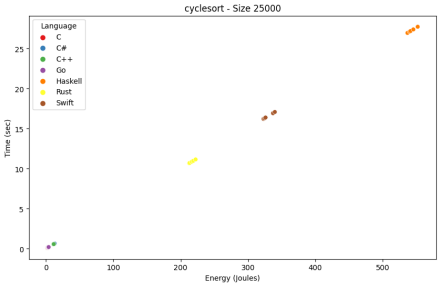


(b) BubbleSort 50000 energy and time

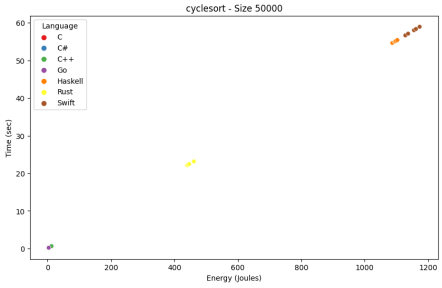


(c) BubbleSort 100000 energy and time

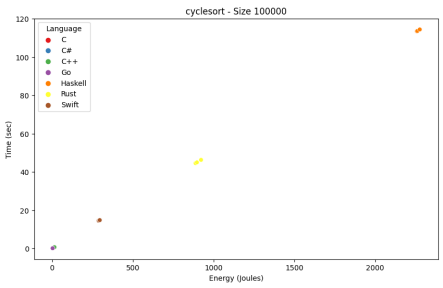
Figure 28. Bubble Sort Chart Energy and Time Compiling with PowerCap



(a) CycleSort 25000 energy and time

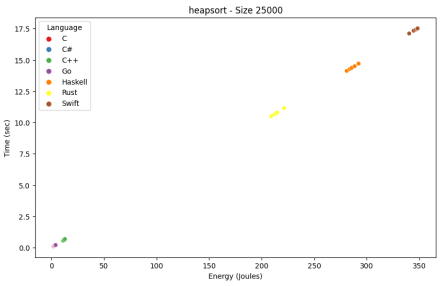


(b) CycleSort 50000 energy and time

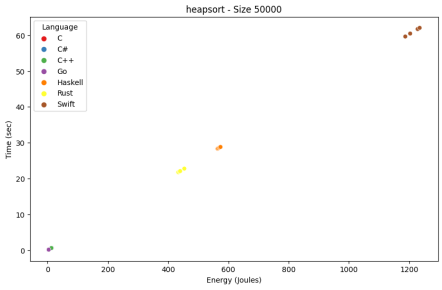


(c) CycleSort 100000 energy and time

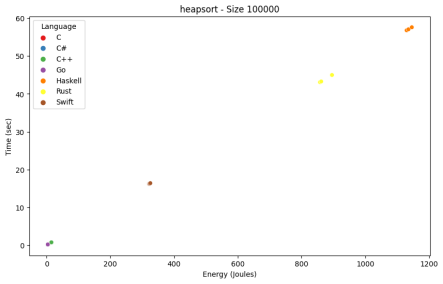
Figure 29. Cycle Sort Chart Energy and Time Compiling with PowerCap



(a) HeapSort 25000 energy and time

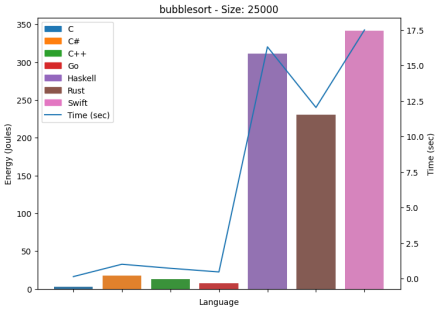


(b) HeapSort 50000 energy and time

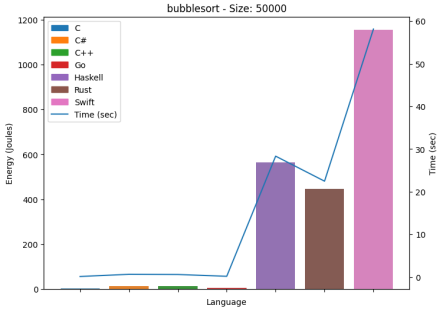


(c) HeapSort 100000 energy and time

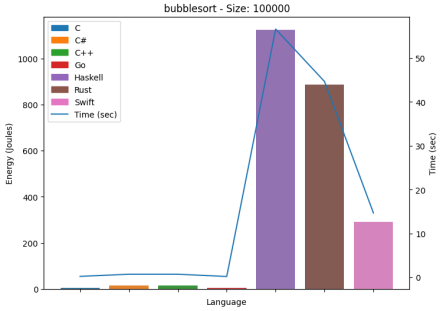
Figure 30. Heap Sort Chart Energy and Time Compiling with PowerCap



(a) BubbleSort 25000 energy and time

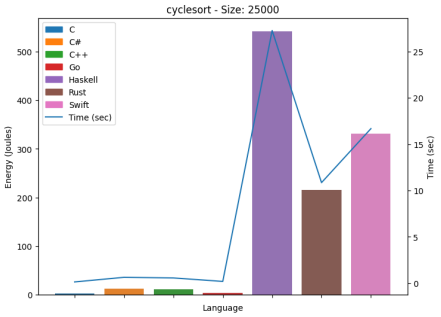


(b) BubbleSort 50000 energy and time

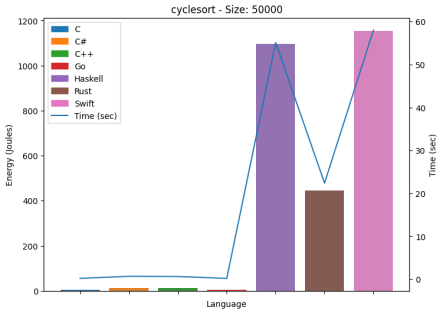


(c) BubbleSort 100000 energy and time

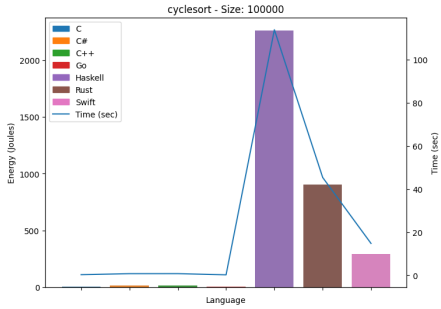
Figure 31. Bubble Sort Plot Bar Energy and Time Compiling with PowerCap



(a) CycleSort 25000 energy and time

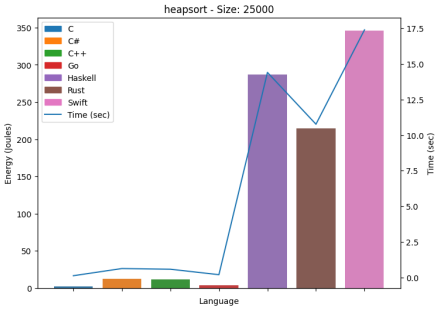


(b) CycleSort 50000 energy and time

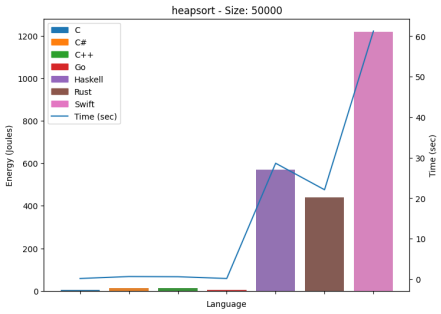


(c) CycleSort 100000 energy and time

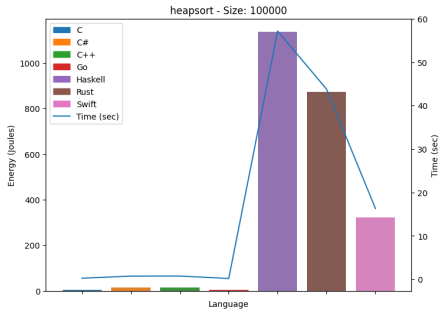
Figure 32. Cycle Sort Plot Bar Energy and Time Compiling with PowerCap



(a) HeapSort 25000 energy and time

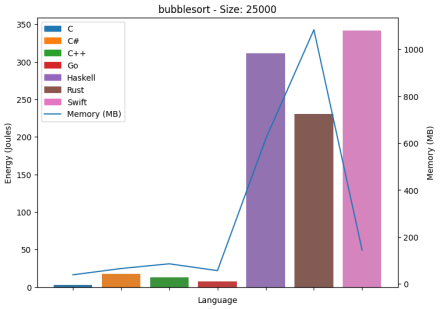


(b) HeapSort 50000 energy and time

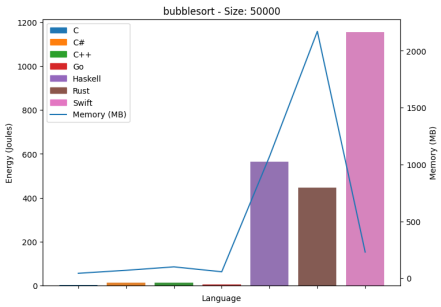


(c) HeapSort 100000 energy and time

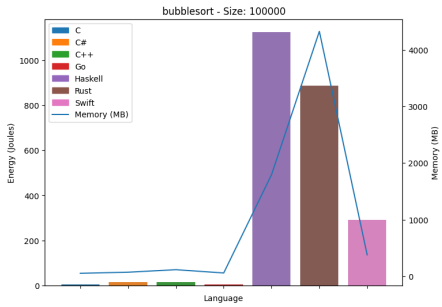
Figure 33. Heap Sort Plot bar Energy and Time Compiling with PowerCap



(a) BubbleSort 25000 energy and memory

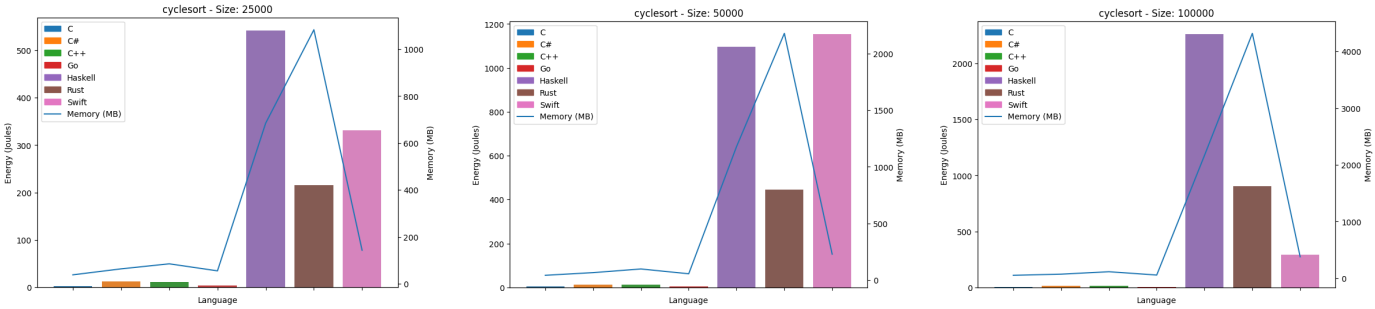


(b) BubbleSort 50000 energy and memory



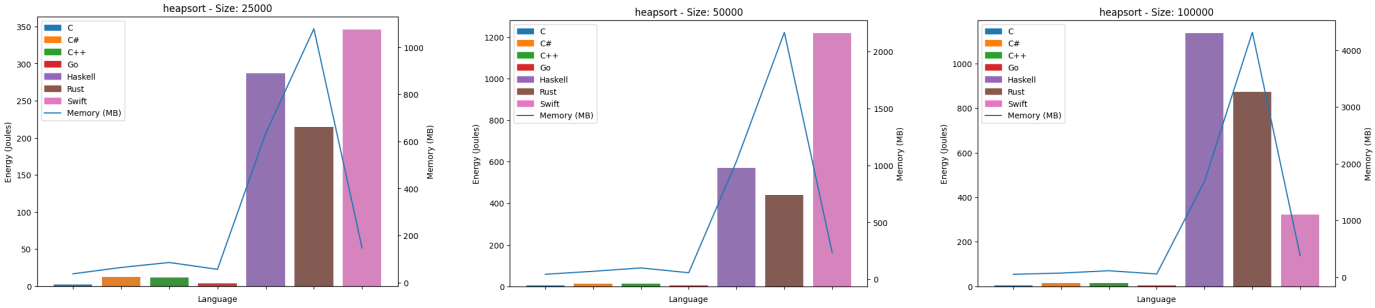
(c) BubbleSort 100000 energy and memory

Figure 34. Bubble Sort Plot Bar Energy and Memory Compiling with PowerCap



(a) CycleSort 25000 energy and memory **(b)** CycleSort 50000 energy and memory **(c)** CycleSort 100000 energy and memory

Figure 35. Cycle Sort Plot Bar Energy and Memory Compiling with PowerCap



(a) HeapSort 25000 energy and memory **(b)** HeapSort 50000 energy and memory **(c)** HeapSort 100000 energy and memory

Figure 36. Heap Sort Plot bar Energy and Memory Compiling with PowerCap

4 Towards a Ranking of Sorting Algorithms

In this section, we will analyze the outcomes of our investigation. Although our primary emphasis is on comprehending energy efficiency in languages, we will also endeavour to grasp the connection between energy, time, and memory. As such, we will try to answer the following questions.

- **RQ1**: Can we build a ranking for the greener languages in terms of execution?
- **RQ2**: Can we build a ranking for the greener languages in terms of compilation?
- **RQ3**: Can we build a ranking for the most efficient sorting algorithm?
- **RQ4**: Is there any relation between execution time, memory peak usage and energetic consumption?

4.1 Ranking of the greener languages in terms of execution

As software applications expand in complexity and scale, understanding the energy efficiency of programming languages becomes essential for optimizing resource consumption and minimizing ecological footprints.

The objective of this analysis is to evaluate and rank programming languages based on their execution efficiency, specifically focusing on their environmental impact in terms of energy consumption. While traditional benchmarks typically prioritize performance, we aim to shed light on a different aspect by emphasizing the green credentials of programming languages. By comparing and contrasting various languages, we can identify those that offer superior execution efficiency and minimize energy usage during program execution.

Therefore, based on the results acquired prior in the average performance of multiple algorithms over multiple mainstream languages, we defined the following ranking in terms of language execution performance:

4.2 Ranking of the greener languages in terms of compilation

One critical aspect that influences the efficiency and sustainability of software development is the compilation process, by evaluating and ranking programming languages based on their compilation efficiency, we can identify the options that promote energy conservation.

The primary objective of this analysis is to assess and compare programming languages in terms of their compilation efficiency by understanding the energy consumption associated with different languages during the compilation process so that developers can make informed decisions that align with sustainable software development practices.

Even though the compilation process can look minimal to the environmental impact due to only being executed one time to generate the binary executable program, the truth is that developers need to compile their programs in order to test them and find multiple bugs, being a constant loop in software development. That way, we can easily understand that the impact of compilation can be very important in this matter, mainly to programs of great dimensions that have a tendency to have multiple bugs or even to be developed during a great period of time, causing a great number of needed compilations.

Furthermore, this analysis aims to investigate the relationship between compilation efficiency, time complexity, and memory utilization in programming languages, gaining insights into the trade-offs and optimizations that can be made to enhance the performance of programming languages during the compilation stage.

Based on the prior results from the average performance of multiple algorithms over multiple mainstream languages, we defined the following racking in terms of language compilation performance:

4.3 Ranking for the most efficient sorting algorithm

Sorting algorithms have a significant impact on the performance and efficiency of various applications. With the ever-increasing size and complexity of data, selecting the most efficient sorting algorithm becomes crucial for optimizing execution time and resource utilization. This analysis aims to compare and rank different sorting algorithms based on their efficiency, providing valuable insights for developers and researchers in selecting the most appropriate algorithm for their specific requirements.

The objective of this analysis is to evaluate and rank sorting algorithms based on their efficiency in terms of time, memory, and overall performance, as well as to explore trade-offs between different sorting algorithms. Efficient sorting algorithms offer the advantage of minimizing the number of comparisons, swaps, and memory usage, resulting in faster and more resource-friendly sorting operations.

Based on the prior results from the average performance of multiple algorithms over multiple mainstream languages, we defined the following racking to determine the most efficient sorting algorithm:

4.4 Execution time, memory peak usage and energetic consumption

Optimizing resource usage is a critical objective to enhance application performance and reduce operational costs. Execution time, memory peak usage and energetic consumption significantly impact software efficiency and so, understanding the intricate relationship between these variables can provide valuable insights for developers, enabling them to make informed decisions regarding resource allocation and energy-efficient programming practices.

Execution time represents the duration it takes for a program to complete its execution, directly influencing user experience and application responsiveness. Memory peak usage refers to the maximum amount of memory utilized

by a program during its execution, affecting system performance and scalability. Energetic consumption measures the amount of energy consumed by the application, highlighting its environmental impact and operational costs. By examining these factors, we aim to uncover potential correlations and trade-offs that can guide developers in optimizing their code for improved efficiency.

Based on the prior results from the average performance of multiple algorithms over multiple mainstream languages, we defined the following possible correlations between these three factors:

4.5 Multi-Criteria Decision Making

To make some valid conclusions one can either look manually at the data and infer what he wants or engage in a systematic approach using Multi-Criteria Decision Making (MCDM) methods. These methods are part of a structured methodology used to evaluate and prioritize alternatives based on multiple criteria or factors, proving a rigorous framework for decision-makers to analyze complex problems and make informed choices and helping them avoid biases and subjectivity.

Therefore, utilizing different methods of Multi-Criteria Decision Making (MCDM) is generally considered superior to comparing data or analyzing data in the form of graphs alone. While analyzing graphs can provide some insights and simplify the decision-making process to some extent, MCDM offers a more systematic and comprehensive approach, using thresholds, preferences, weights or even a prioritization rank. This is essential for this study because the use of this method can surpass the necessity of anyone: Sometimes it is desirable to give more weight to the execution time. However in another situation, it can be desirable to give more weight to energy consumption.

For this study, we made available in our website the following methods for the user to test different approaches for either knowing which algorithm or programming language is better when certain conditions met:

- Pareto;
- Promethee;
- Weighted Sum;
- Electre Iv/Is.

4.5.1 Pareto

4.5.2 Promethee

4.5.3 Weighted Sum

Simple multicriteria decision-making technique. It involves assigning weights to each criterion and evaluating alternatives based on these weights. On our website, an analysis can be made by giving as an input a weight from 0 to 10 to each of the following entries: energy consumption, memory consumption and time. With the language study case, one can also give weight to the IEE Score from 2022 (completar com a fonte)

4.5.4 Electre Iv/Is

5 Validity

Our research had multiple goals: firstly, and the main one, was to analyze the energetic consumption of the execution and compile process of sorting algorithms in multiple languages, mainly, the most mainstream ones. Although due to the tool we used to generate the code, GPT-3.5 [4], we also were curious to know about the capabilities of this tool to generate green and efficient code in numerous languages and in multiple paradigms. In this section, we present some threats to the validity of our results divided into four categories [12]: conclusion validity, construct validity, external validity and internal validity.

5.1 Conclusion Validity

With the obtained results, it's very clear that different programming paradigms and even languages within the same paradigms show completely different results between each other, and even between sorting algorithms. With ones being excellent in ones (p.e. Haskell in Bubble Sort), but extremely poor in another (p.e. Haskell in Cycle Sort). We can also see that the programs which present the most energetic consumption in some algorithms aren't, always, the ones with the worst consumption in compiling, or vice versa.

Unfortunately, due to the computer in which we do the tests being from a recent Intel generation, RAPL wasn't able to collect data about the DRAM energetic consumption, so we only worked with the Package consumption, calling it Energy in this paper. This way, the memory peak usages show only which algorithms or languages need more memory to execute, and we can't be sure about the energetic consumption made by those usages. We could also check that the memory usage presents a great variation between algorithms and languages, not being directly correlated with the energy consumption (p.e. for C#, the values of energetic consumption are very low when compared with the memory usage of its algorithms).

By applying the PowerCap, as expected, we could see an increase in the execution times of the algorithms in the multiple languages, in general.

We grouped the languages the way we felt was more appropriate to do it and inspired in [24], although the data is available in our GitHub repository and on our website, so everyone can change this grouping very easily.

5.2 Construct Validity

Another of the goals of our research was also to analyze the capabilities of GPT-3.5 [4] in generating correct, efficient and green programs for our questions. With our results, we can easily, and comparing with previous works under

similar conditions [14, 24], check that GPT-3.5 presents very unstable solutions to each language, giving great programs for some sorting algorithms (p.e. Bubble Sort in Haskell), but horrible ones to another (p.e. Cycle Sort in Haskell).

Despite this, all the programs were generated by the same source and under the same prompts, providing us with tools not only to check the efficiency of programming languages but also the efficiency of code generators like GPT-3.5.

In order to get a more reliable source of sorting algorithms if the goal of the research was only to analyze them and their efficiency, other strategies could have been used, like the ones in [24]. Yet, given our objectives, we consider that we got interesting data to answer all of our questions.

5.3 Internal Validity

When measuring the energy consumption of the various sorting algorithms in multiple programming languages, many factors, along with the different implementations and programming languages, can affect the results, like energy spikes, or specific versions of compilers or VMs. To avoid that, we executed each algorithm, with each input size in each language, 5 times. The input arrays were generated in a Python script and were hard-coded in each algorithm, to have the same values in the same positions. Later, when analyzing our data, the entries with negative values (caused by overflows in RAPL measures) were removed.

The used energy measurement tool was also proven to be very accurate in previous works [14, 24].

The obtained results are very consistent and considered reliable.

5.4 External Validity

It is expected that in the future, our source of code and even the compilers, interpreters or VMs utilized will improve a lot, so it is very plausible to repeat this research in the future with these new tools and expect some different results that should not vary a lot from the ones obtained in this one.

All the info utilized in this study is very well documented and can easily be replicated in the future, so we believe our results can be further generalized, and other researchers can replicate our methodology for future work.

6 Related Work

The work presented in this paper extends the work presented by [24], where the stoppage time between different algorithms was hard-coded and there weren't used any of the [6], [3] or [2] elements.

To give more information about greener software, we also extend the work in [24] by measuring, not only the consumption of the program's execution but also of its compiling process. With this, we allow the developers to have a richer knowledge about what software is green, once the debugging process will, most of the time, require multiple compile processes of the program, resulting in a great energetic consumption as well, and not only on its execution.

We also support our work in [4], which allowed us to test its capabilities and check one of the supposed technologies of the future.

7 Conclusions

References

[1] 2015. Intel's Running Average Power Limit. https://software.intel.com/en-us/articles/intel-power-governor

[2] 2017. sys_resource man pages. https://man7.org/linux/man-pages/man0/sys_resource.h.0p.html Last Accessed: 2023-05-27.

[3] 2021. lm-sensors on GitHub. https://github.com/lm-sensors/lm-sensors Last Accessed: 2023-05-27.

[4] 2023. Chat GPT. https://chat.openai.com/ Last Accessed: 2023-05-27.

[5] 2023. Random Library in Python. https://docs.python.org/3/library/random.html Last Accessed: 2023-05-27.

[6] 2023. Raplcap on GitHub. https://github.com/powercap/raplcap Last Accessed: 2023-05-27.

[7] 2023. Sorting Algorithms. https://www.geeksforgeeks.org/sorting-algorithms/ Last Accessed: 2023-05-27.

[8] 2023. The Computer Language 23.03 Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html Last Accessed: 2023-05-27.

[9] Sarah Abdulsalam, Ziliang Zong, Qijun Gu, and Meikang Qiu. 2015. Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. 1–8. https://doi.org/10.1109/IGCC.2015.7393699

[10] Reyhaneh Jabbarvand Behrouz, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. 2015. EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 8–14. https://doi.org/10.1109/GREENS.2015.9

[11] Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 49–60.

[12] Thomas D Cook and D T Campbell. 1979. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin.

[13] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. 2017. Products Go Green: Worst-Case Energy Consumption in Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A (SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3106195.3106214

[14] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages *(SBLP '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3125374.3125382

[15] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. 40, 3 (jan 2012), 13–17. https://doi.org/10.1145/2425248.2425252

[16] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 92–101. https://doi.org/10.1109/ICSE.2013.6606555

[17] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/2884781.2884869

[18] Ding Li and William G. J. Halfond. 2014. An Investigation into Energy-Saving Programming Practices for Android Smartphone App Development *(GREENS 2014)*. Association for Computing Machinery, New York, NY, USA, 46–53. https://doi.org/10.1145/2593743.2593750

[19] Ding Li, Shuai Hao, Jiaping Gui, and William G.J. Halfond. 2014. An Empirical Study of the Energy Consumption of Android Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 121–130. https://doi.org/10.1109/ICSME.2014.34

[20] Luis Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor Filho, Gilberto Melfe, and João Paulo Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 517–528.

[21] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. 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)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/2597073.2597085

[22] Wellington Oliveira, Renato Oliveira, and Fernando Castor. 2017. A Study on the Energy Consumption of Android App Development Approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 42–52. https://doi.org/10.1109/MSR.2017.66

[23] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 238–240. https://doi.org/10.1109/ICSE-C.2017.80

[24] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. 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 (SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[25] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS '16)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/2896967.2896968

[26] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions about Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2597073.2597110

[27] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding Energy Behaviors of Thread Management Constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 345–360. https://doi.org/10.1145/2660193.2660235

[28] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (2012), 20–27. https://doi.org/10.1109/MM.2012.12

[29] Cagri Sahin, Lori Pollock, and James Clause. 2014. How Do Code Refactorings Affect Energy Usage? *(ESEM '14)*. Association for Computing Machinery, New York, NY, USA, Article 36, 10 pages. https://doi.org/10.1145/2652524.2652538

[30] Anne E. Trefethen and Jeyarajan Thiyagalingam. 2013. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science* 4, 6 (2013), 444–449. https://doi.org/10.1016/j.jocs.2013.01.005 Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[31] Tomofumi Yuki and Sanjay Rajopadhye. 2014. Folklore Confirmed: Compiling for Speed $$=$$Compiling for Energy. In *Languages and Compilers for Parallel Computing*, Călin Caşcaval and Pablo Montesinos (Eds.). Springer International Publishing, Cham, 169–184.